UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

INGENIERÍA TÉCNICA DE TELECOMUNICACIÓN
TELEMÁTICA

PROYECTO FINAL DE CARRERA

IMPLEMENTACIÓN DE UN SERVICIO DE FILTRADO Y AGREGACIÓN PARA COMUNIDADES EN TWITTER

Autor: **Marta García González**
Tutor: **Norberto Fernández García**

Julio de 2012

Agradecimientos

Creí que nunca llegaría el día en el que pudiese escribir los agradecimientos de mi proyecto fin de carrera, pero ese día al fin ha llegado. Tras varios meses de esfuerzo puedo decir que por fin he concluido este trabajo en el que he puesto tanta ilusión y empeño.

Quiero agradecérselo especialmente a mis padres. Gracias por la educación y los valores que me habéis enseñado desde pequeña. Gracias por haberme transmitido siempre el interés y la curiosidad por las cosas. Sin vuestro apoyo, cariño, y vuestra compresión constante, todo esto hubiera sido una quimera.

A mi hermano y a Isabel, por estar siempre ahí cuando les necesito, por escucharme y darme buenos consejos.

A mi «hermana» Elena, por ser mi mejor amiga y una persona maravillosa. Y a mis biólogos favoritos, Fito, María, Ricardo y Shaky.

A mis compañeros de universidad, especialmente a Alvarito, Ana, Bea, Carlos, Guille, Jan, Jorge, Juanal, Marce, Pao, Raquel y San. Empezasteis siendo compañeros de aula pero terminasteis siendo grandes amigos con los que sigo y espero seguir teniendo contacto siempre. Hemos compartido sufrimiento y agobios en prácticas y exámenes, pero también hemos pasado innumerables buenos momentos juntos. Sois muy grandes.

Y por último, muchas gracias a Luis por la idea y a mi tutor de proyecto, Norberto. Sin su paciencia durante estos meses y su tiempo invertido resolviendo dudas cuando lo he necesitado, esto no hubiera sido posible.

No te rindas, por favor no cedas,
aunque el frío queme,
aunque el miedo muerda,
aunque el sol se esconda
y se calle el viento.
Aún hay fuego en tu alma,
aún hay vida en tus sueños.
Porque la vida es tuya,
y tuyo también el deseo,
porque cada día es un comienzo nuevo,
porque esta es la hora y el mejor momento.

Fragmento de *«No te rindas»*, de Mario Benedetti

Lista de acrónimos

- **AJAX** - *Asynchronous JavaScript And XML*
- **API** - *Application Programming Interface*
- **CRUD** - *Create, Read, Update and Delete*
- **CSS** - *Cascading Style Sheets*
- **DOM** - *Document Object Model*
- **IDE** - *Integrated Development Environment*
- **JDK** - *Java Development Kit*
- **DM** - *Direct Message*
- **DRY** - *Don't Repeat Yourself*
- **GORM** - *Graphical Object Relationship Modeller*
- **GSP** - *Groovy Server Pages*
- **HQL** - *Hibernate Query Language*
- **HTML** - *HyperText Markup Language*
- **HTTP** - *Hypertext Transfer Protocol*
- **IP** - *Internet Protocol*
- **JDBC** - *Java Database Connectivity*
- **JEE** - *Java Platform, Enterprise Edition*
- **JSE** - *Java Platform, Standard Edition*
- **JSON** - *JavaScript Object Notation*
- **JSP** - *Java Server Pages*
- **JTA** - *Java Transaction API*
- **OAuth** - *Open Authorization*
- **PC** - *Personal Computer*
- **REST** - *Representational State Transfer*
- **RSS** - *Really Simple Syndication*
- **RT** - *Retweet*
- **SMS** - *Short Message Service*
- **STS** - *SpringSource Tool Suite*

- **TL** - *Timeline*
- **TT** - *Trending Topic*
- **URL** - *Uniform Resource Locator*
- **W3C** - *World Wide Web Consortium*
- **XML** - *Extensible Markup Language*

Resumen

El fenómeno de las redes sociales ha supuesto el comienzo de una nueva era marcada por las comunicaciones interpersonales en tiempo real en detrimento del uso de otros medios de comunicación tales como el correo electrónico, los foros, los SMS, etc.

Una de las redes sociales de más éxito y repercusión actualmente es *Twitter*, un servicio gratuito de *microblogging* que permite a sus usuarios publicar mensajes de texto cuya longitud máxima es de 140 caracteres.

Twitter está basado en un formato muy simple, tanto en su diseño como en su utilización, y quizás en eso radique su éxito mundial. Los usuarios que lo utilizan son de diversa índole y los usos que se le dan son múltiples. Van desde una utilización meramente personal, hasta una utilización enfocada al ámbito empresarial.

Una de las funcionalidades que *Twitter* ofrece a un usuario, es poder seguir a otros usuarios que publiquen información que sea relevante para él. Así, podrá leer desde su cuenta noticias y otros datos de interés que publiquen sus *followings*[1]. Si el usuario desea que la información que le ha parecido interesante sea compartida también con sus *followers*[2], éste deberá publicar en su cuenta de *Twitter* esos mensajes relevantes que han sido escritos por otros. Para ello tendrá que hacer un *retweet* (RT)[3] de cada uno de estos *tweets* relevantes.

Muchas veces, es tanta la cantidad de información publicada por los *followings*, que es difícil que un usuario sea capaz de mantenerse al día y leer todo lo que éstos publiquen. Puede que algunos mensajes de interés pasen desapercibidos debido a que no hay filtros que separen la información que resulta relevante de la que no lo es. Para evitar que esto suceda, se ha creado un servicio de filtrado y agregación de cuentas de *Twitter* a partir de temáticas definidas a través de *hashtags*[4].

Para la configuración de tal servicio, un usuario de *Twitter* que tome el rol de administrador en el sistema, a través de una interfaz web implementada, será el encargado de conformar una comunidad compuesta por una serie de usuarios de *Twitter* y una cuenta robot. El objetivo es que el administrador tenga el control sobre lo que en la cuenta robot se publique. Con ello se consigue que en vez de tener que seguir a muchos usuarios que publiquen *tweets* con *hashtags* de las mismas temáticas, sea necesario seguir únicamente a la cuenta robot, puesto que ésta publicará los contenidos ya filtrados de todos los usuarios de *Twitter* pertenecientes a la comunidad.

[1] usuarios a los que sigue otro usuario de *Twitter*

[2] seguidores de un usuario de *Twitter*

[3] si un usuario hace un RT de un *tweet* de otro usuario de *Twitter*, es como si reenviase dicho *tweet* a todos sus *followers*

[4] palabra precedida del caracter '#' que se utiliza en los *tweets* para poder categorizarlos por temas

Abstract

Social networks phenomenon has involved the beginning of a new era characterized by interpersonal communications in real time and by the decrease in the use of other communication methods such as SMS, mail, internet forums...

Nowadays, *Twitter* is one of the social networks with more success and impact. *Twitter* is a free *micro-blogging* service where users can publish messages called *tweets* with a maximum size of 140 characters.

Twitter has a simple format, and both its usage and design are also simple. Maybe its success lies in the simplicity. *Twitter* is used both for personal and business uses. Many companies use this service for promotions, sales, publicity, to ask questions to their clients...

One of the main features *Twitter* provides to its users is to follow other users whose published information is interesting. In this way, the user will be able to read in his account notices and other interesting publications which have been published by other *Twitter* users that are followed by him (*followings*). If the user wishes the relevant information be shared with his *followers*, he must publish in his *Twitter* account all the relevant messages (relevant *tweets*) which have been written by others. For this, he will have to do *retweet* (RT) of these interesting *tweets*.

Sometimes, the quantity of information published by the users that are followed is enormous. Due to this it is very difficult for a user to read all the *tweets* that were published by their *followings*. It may happen that some messages of interest get unnoticed because there are not filters that split the relevant information from the irrelevant one. To avoid this problem, it has been created a filtering and aggregation service of *Twitter* accounts through topics defined by *hashtags*.

To configure this service, a *Twitter* user who will have the admin role, will form a community with several *Twitter* users and one *Twitter* account that will be a robot account. The admin will have the control over the robot account and will decide all the *tweets* to be published in this account. In this way, instead of following many users that publish *tweets* with *hashtags* about the same topics, it will only be necessary to follow the robot account, because this account will publish all relevant *tweets* which were published by users that belong to the community.

Índice general

Indice de figuras

Capítulo 1

Motivación y contexto del proyecto

1.1. Motivación

En los últimos años, la sociedad se ha modernizado notablemente y ha experimentado un cambio vertiginoso en lo que a los medios de comunicación interpersonal se refiere. Hace veinte años, la comunicación entre individuos que querían mantener el contacto se realizaba mediante carta o vía telefónica. Hace algo más de una década, se empezaron a imponer los teléfonos móviles y el correo electrónico. A día de hoy, en cambio, prima la comunicación a través de redes sociales tales como *Facebook*, *Twitter*, etc.

Nuevas tecnologías como los *smartphones* aunadas con los precios ofertados por compañías telefónicas en tarifas de datos para teléfonos móviles, han propiciado que las redes sociales sigan incrementando su ya consolidado éxito entre la sociedad. Ahora la gente ya no solo puede acceder a estos servicios desde su casa, sino también desde sus dispositivos móviles. Es por eso que las redes sociales, además de jugar un factor muy importante a nivel de comunicación interpersonal, también juegan un factor importante a nivel informativo o publicitario. Un buen ejemplo de este fenómeno emergente es *Twitter*, la red social en la que se sustenta este proyecto.

Twitter[1] es un servicio de *microblogging* que surgió en el año 2006, creado por Jack Dorsey. La simplicidad de su diseño y la facilidad de uso lo hacen muy atractivo al usuario, quizás en ello radique su éxito mundial. Se estima que tiene más de 200 millones de usuarios y que genera más de 65 millones de *tweets* y más de 800.000 peticiones de búsqueda diariamente. Estos datos han hecho que se le conozca como el SMS de Internet.

Twitter permite publicar mensajes de texto plano, denominados *tweets*, con una limitación de 140 caracteres. Un usuario con cuenta en *Twitter* puede seguir a otros usuarios; puede ser seguido; puede enviar mensajes directos (DMs), que son mensajes privados dirigidos a un usuario en concreto; y puede hacer *retweets* (RTs). Los *tweets* pueden contener menciones, o lo que es lo mismo, estar dirigidos a otros usuarios en concreto; y también pueden contener *hashtags*, que son palabras precedidas por el caracter `'#'` que sirven para etiquetar los *tweets* y facilitar las búsquedas por temáticas de los mismos.

Tal y como se ha mencionado, existe un amplio abanico de posibles usos de *Twitter* (empresarial, personal, informativo, etc.). Es por ello por lo que surge la idea de implementar este proyecto.

El principal interés de un usuario de *Twitter*, además de publicar su propia información, es seguir a otros que publiquen contenido relevante para él. De esta manera, el usuario podrá leer desde su página principal todas las noticias y otros datos de interés que publiquen sus *followings*. Si el usuario desea que la información que le ha parecido interesante sea compartida también con sus *followers*, éste deberá publi-

car en su cuenta de *Twitter* esos mensajes relevantes que han sido escritos por otros. Para ello tendrá que hacer un RT de los mismos.

Muchas veces, es tanta la cantidad de información publicada diariamente por los *followings* que un usuario sigue, que es difícil que éste sea capaz de mantenerse al día y leer todos los *tweets* de su *timeline (TL)*[1]. Puede que algunos mensajes de interés pasen desapercibidos. Por ello, surge la idea de implementar un **servicio de filtrado y agregación de cuentas de *Twitter* a partir de temáticas definidas a través de *hashtags***.

La idea es que con el sistema desarrollado, un usuario de *Twitter* que tome el rol de administrador, sea el encargado de conformar una comunidad compuesta por una cuenta robot (que será una cuenta de *Twitter*) y por un conjunto de usuarios de *Twitter* agregados a la misma. La cuenta robot publicará RTs de *tweets* que contengan determinados *hashtags* y que hayan sido publicados por alguno de los usuarios agregados.

De este modo se consigue que en la cuenta robot se publiquen únicamente los *tweets* que contengan temas de relevancia para una comunidad. Así, en vez de tener que seguir en *Twitter* a muchos usuarios que publiquen *tweets* con *hashtags* de las mismas temáticas, será necesario únicamente seguir a la cuenta robot.

La aplicación se utilizará para seguir los canales *Twitter* de los miembros de la Asociación de Telemática `ATEL`, y filtrar/agregar *tweets* de interés para la comunidad.

1.2. Objetivos

El objetivo principal que se definió al principio del proyecto, fue crear una aplicación web en la que diversos usuarios se pudiesen dar de alta para configurar cuentas de *Twitter* que funcionasen como cuentas robots.

Se considera cuenta robot a una cuenta de *Twitter* que se actualiza automáticamente, sin necesidad de que el usuario interactúe manualmente para publicar *tweets* o RTs.

El escenario propuesto fue el que se muestra en la figura 1.1. Como se puede observar, cada administrador que se da de alta en el sistema debe asociar a su cuenta una cuenta robot, así como un conjunto de cuentas de usuarios de *Twitter*. Es importante destacar que **un administrador puede administrar una única cuenta robot** y **una cuenta robot puede ser gestionada por un solo administrador**.

El administrador también tendrá que configurar, a través de una serie de comandos, los *hashtags* que son relevantes para él. Así asociará a cada uno de los usuarios determinados *hashtags*. Por lo tanto, un *tweet* será relevante para una cuenta robot dependiendo de quién sea su autor y de cuáles sean los *hashtags* que contiene. Si un *tweet* es relevante, la cuenta robot hará RT del mismo.

Siguiendo el ejemplo de la figura 1.1 para la cuenta robot `cuentaRobot1`:

1. Suponer que su administrador asociado, `admin1`, ha configurado que los *tweets* publicados por el `user11` que contengan el *hashtag* `#hashtag1` son relevantes para él.

2. Suponer que el `user11` actualiza su estado y publica un *tweet* cuyo contenido es "`Este es un tweet de ejemplo que incluye #hashtag1`".

[1] página de inicio de *Twitter* de un usuario. Muestra los *tweets* de los *followings* del usuario en orden cronológico inverso

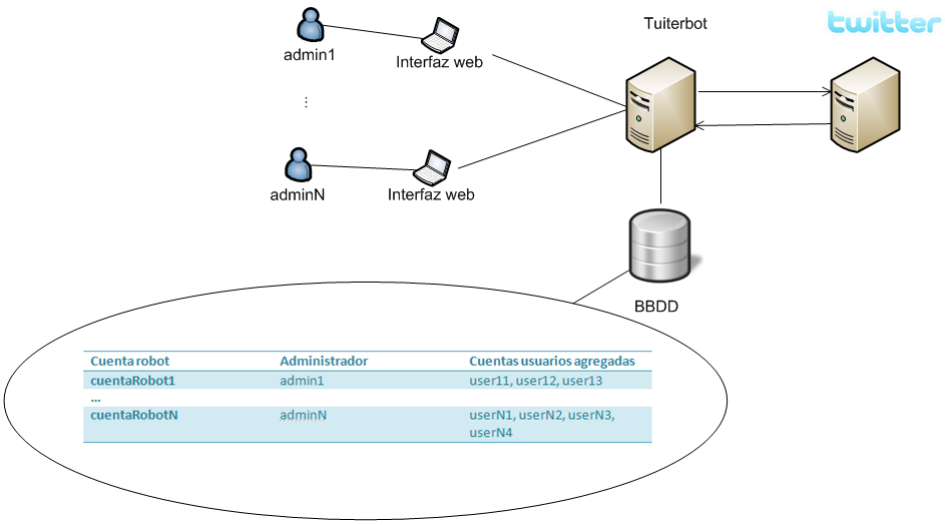


Figura 1.1: Esquema de la aplicación Tuiterbot

La consecuencia de esto es que en cuanto el sistema compruebe que `user11` ha publicado un *tweet* relevante para `cuentaRobot1`, ésta hará RT automáticamente de dicho *tweet*.

1.3. Estructura de la documentación

El presente documento está compuesto por siete capítulos y tres apéndices. A lo largo de todos ellos se exponen detalladamente los aspectos considerados relevantes durante el desarrollo del proyecto.

A continuación se ofrece una breve síntesis del contenido de cada uno de estos capítulos y apéndices.

- **Capítulo 1. Motivación y contexto del proyecto**. Se comienza haciendo una breve introducción de los aspectos que motivaron el desarrollo del proyecto y de los objetivos propuestos inicialmente para llevar a cabo su realización.
- **Capítulo 2. Estado del arte**. En este capítulo se detallan las distintas tecnologías que fueron utilizadas para la implementación del proyecto. Entre ellas cabe destacar el lenguaje de programación *Groovy*, el *framework Grails*, *Quartz*, etc.
- **Capítulo 3. Requisitos de diseño**. En el tercer capítulo se indican los requisitos de diseño que se fijaron antes de comenzar el desarrollo de la aplicación.
- **Capítulo 4. Diseño de alto nivel**. En el capítulo de diseño de alto nivel, tal como indica su título, se hablará del diseño de alto nivel del sistema. Cumpliendo con los requisitos de diseño enumerados en el anterior capítulo, se detallarán los distintos casos de uso que se pueden dar al utilizar un usuario la aplicación.
- **Capítulo 5. Implementación**. En este capítulo se especificará más en detalle cómo se ha implementado cada una de las partes que conforman el código. Su estructura de ficheros, el diseño de base de datos, las capas de la aplicación, etc. Para mostrar esto, se incluirán determinadas partes del código.
- **Capítulo 6. Pruebas**. En este capítulo se detallan, una por una, todas las pruebas realizadas para comprobar el correcto funcionamiento del software implementado. Las pruebas se realizan con el

objetivo de comprobar que el comportamiento es el deseado para cada uno de los casos de uso que se pueden dar.

- **Capítulo 7. Conclusiones**. En el último capítulo se describen las conclusiones a las que se ha llegado durante la realización de la aplicación. Se comentarán los obstáculos que han surgido durante el desarrollo software de la aplicación y también los resultados obtenidos al final de la etapa de desarrollo.

- **Apéndice A. Manual de instalación**. En este apéndice se detallan los pasos que debe realizar un usuario para instalar la aplicación. La instalación se explica desde dos puntos de vista. Por un lado, desde el punto de vista de un desarrollador que vaya a hacer alguna modificación en la aplicación a través del código fuente de la misma, y por otro lado, desde el punto de vista de un usuario que quiera ejecutar la aplicación desde su PC.

- **Apéndice B. Manual de usuario**. En este apartado se explican al usuario las instrucciones de uso de la aplicación. Se detallan las funcionalidades disponibles a través de la interfaz web, los pasos a seguir para configurar una cuenta robot, etc. Las explicaciones irán acompañadas de un ejemplo práctico.

- **Apéndice C. Planificación**. Este último apéndice describe la historia del proyecto: planificación, distribución temporal, costes, y otros detalles relevantes surgidos durante su desarrollo.

Capítulo 2

Estado del arte

El objetivo principal de este capítulo es detallar las distintas tecnologías y herramientas que se han empleado para la realización del proyecto.

Se comenzará hablando del lenguaje de programación utilizado, *Groovy*, así como del *framework Grails*, herramientas utilizadas para la elaboración del código fuente de la aplicación que es objeto de estudio. También se hablará de la API de *Twitter*, fundamental para acceder a la información de la red social en cuestión, y de *Quartz*, librería utilizada para programar las tareas del *bot*.

2.1. Groovy

2.1.1. Introducción

Groovy[2][3][4][5] es un lenguaje de programación orientado a objetos que está implementado sobre *Java*, por ello, ambos lenguajes guardan muchas similitudes entre sí. Una de las grandes diferencias radica en que *Groovy* puede usarse dinámicamente como un lenguaje de *scripting*.

La primera versión de *Groovy*, la 1.0, apareció el 2 de enero de 2007. Tras varias versiones *beta* y algunas *release*, el 7 de diciembre de 2007 apareció la versión 1.1, que finalmente fue renombrada a 1.5 con el fin de dejar constancia de los grandes cambios sufridos con respecto a la primera versión. En diciembre de 2009 se publicó la versión 1.7. Actualmente, la última versión estable es la 1.8, pero existen versiones beta hasta de la 2.0.

Groovy ofrece unas características muy similares a las de otros lenguajes de programación tales como *Python*, *Perl* y *Ruby*. Aún así, al que más se parece en cuanto a sintaxis es a *Java*. Ambos comparten el mismo modelo de datos, de hilos y de seguridad. A través de *Groovy* se pueden usar ficheros de código *Java*, lo que significa que se puede acceder a todas las API existentes en *Java*. El *bytecode* que se genera en el proceso de compilación en *Groovy*, es totalmente compatible con el que se genera en *Java* para la máquina virtual (JVM), por lo que puede usarse directamente en cualquier aplicación *Java*. Todo esto hace que sea muy sencillo para un programador que esté familiarizado con *Java*, familiarizarse con *Groovy*. Además, a esto hay que unir la ventaja de que el esfuerzo de aprendizaje es mucho menor que si se intenta pasar directamente a otros lenguajes como *Jython* o *JRuby*, que también generan *bytecode* para la JVM.

2.1.2. Instalación

Para instalar *Groovy*, se debe descargar de la página oficial (`http://groovy.codehaus.org/`) la última versión disponible. Suponiendo que se instala en un entorno *Windows*:

- Descomprimir el `.zip` descargado con la última versión de *Groovy* en el directorio en el que se quiera instalar la distribución. Por ejemplo, si se usa el directorio por defecto, éste será `C:\groovy-1.8.6\bin`.
- El siguiente paso será configurar las variables de entorno JAVA_HOME y GROOVY_HOME. Es necesario que una distribución *Java* esté instalada previamente. En JAVA_HOME se debe indicar la ruta en la que se encuentre instalado el JDK, por ejemplo, `C:\Program Files\Java\jdk1.6.0_27`. En GROOVY_HOME se debe indicar la ruta raíz de *Groovy*, que en el caso del ejemplo es `C:\groovy-1.8.6`.

2.1.3. Características

Algunas de las características que distinguen a *Groovy* son el tipado estático y dinámico, las *closures*, la sobrecarga de operadores, la sintaxis nativa para la manipulación de listas y *maps*, el soporte nativo para expresiones regulares, la iteración polimórfica, y las expresiones embebidas dentro de *strings*.

Debido a estas características, si se compara un fragmento de código *Java* con uno *Groovy*, se puede apreciar que éste último tiene una sintaxis mucho más compacta.

Por ejemplo, el contenido del *main* en el siguiente fragmento de código en *Java*,

```
public class StdJava
{
  public static void main(String argv[])
  {
    for (String it : new String [] {"Spielberg", "Scorsese",
    "Fincher"})
      if (it.length() <= 4)
        System.out.println(it);
  }
}
```

en *Groovy* puede expresarse en una sola línea:

```
["Spielberg", "Scorsese", "Fincher"].findAll{
it.size() <= 4}.each{println it}
```

La mayor parte de cosas que se hacen en *Java* pueden hacerse en *Groovy*:

- Separación de código en paquetes.
- Definición de clases (excepto clases internas) y métodos.
- Uso de estructuras de control (excepto el bucle `for (;;)`).
- Asignaciones, operadores y expresiones.
- Gestión de excepciones.

- Uso de literales.
- Instanciación, referencia a objetos y llamadas a métodos.
- Los comentarios se crean igual, con `//` o con `/**/`.

Además, *Groovy* incorpora mejoras que hacen que el código sea más breve y compacto:

- Facilidad en el manejo de objetos mediante nuevas expresiones y sintaxis.
- Distintas formas de declaración de literales.
- Nuevas estructuras de control de flujo más avanzado.
- Nuevos tipos de datos, con operaciones y expresiones específicas.
- Todo se trata como un objeto.
- No es necesario utilizar punto y coma `';'` al final de una línea de código.
- Se pueden escribir *scripts* en los que no hay clases (en realidad sí que las hay, pero no es necesario declararlas).
- La palabra reservada `def`, permite declarar una variable de tipo dinámico. Aunque también es posible declarar variables de tipo estático indicando un tipo concreto.
- `print` y `println` son equivalentes a `System.out.print` y a `System.out.println`, respectivamente.
- El parámetro a las funciones `print` y `println`, en *Groovy*, no va entre paréntesis. Es opcional ponerlos.

Punto y coma opcional

Como ya se ha mencionado, el caracter punto y coma `';'` es opcional en *Groovy*. Únicamente debe ser utilizado cuando en una misma línea de código se escriben varias sentencias, para diferenciarlas. Por ejemplo:

```
println "sentencia 1"; println "sentencia 2"
```

Paréntesis opcionales

Los paréntesis, al llamar a cualquier método al que se le pasen parámetros, son opcionales. Si el método no tiene parámetros, es obligatorio ponerlos. En caso contrario el compilador no sabría si se trata de un método o de una variable. Ejemplo:

```
void metodo(String s1, String s2) {
    println s1 + s2
}
metodo "uno", "dos"
```

Return opcional

Si un método devuelve algo, la sentencia `return` también es opcional. Si no se indica, se devolverá el resultado devuelto por la última línea del método. Por ejemplo:

```
void suma(int a, int b) {
    a + b
}
```

Es necesario indicar el `return` de manera explícita cuando, por ejemplo, se necesite devolver un valor desde dentro de un bucle. En el caso de que la última sentencia de un método o de un *script* no devuelva ningún valor (por ejemplo, que la última línea sea una sentencia `println`), el método devolverá `null`.

Declaración de tipos opcional

En *Groovy*, la declaración de tipos es opcional. La palabra reservada `def` permite declarar una variable de tipo dinámico, aunque también es posible declarar variables de tipo estático indicando un tipo concreto.

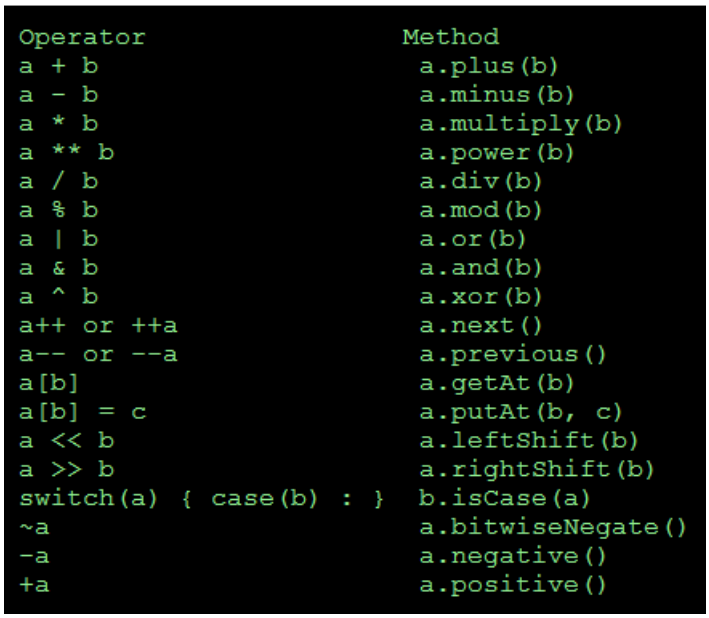
Ejemplo:

```
class MiClase {
    def x = "Esto es un String"

    void metodo() {
        println x.class
        x = 12
        println x.class
    }
}

new MiClase().metodo()
```

Importaciones automáticas

Groovy importa por defecto determinados paquetes, así que no es necesario que éstos sean importados de manera explícita. Estos paquetes y clases son:

- `groovy.lang`
- `groovy.util`
- `java.lang`
- `java.util`
- `java.net`
- `java.io`
- `java.math.BigInterger`
- `java.math.BigDecimal`



Operator	Method
a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.multiply(b)
a ** b	a.power(b)
a / b	a.div(b)
a % b	a.mod(b)
a | b	a.or(b)
a & b	a.and(b)
a ^ b	a.xor(b)
a++ or ++a	a.next()
a-- or --a	a.previous()
a[b]	a.getAt(b)
a[b] = c	a.putAt(b, c)
a << b	a.leftShift(b)
a >> b	a.rightShift(b)
switch(a) { case(b) : }	b.isCase(a)
~a	a.bitwiseNegate()
-a	a.negative()
+a	a.positive()

Figura 2.1: Sobrecarga de operadores

Sobrecarga de operadores

Algo muy interesante en *Groovy* es la sobrecarga de operadores, que hace que se pueda trabajar de manera más sencilla con `Maps`, `Collections` y otras estructuras de datos. Los operadores se asignan a métodos para hacer posible esta funcionalidad (ver figura 2.1).

Con el siguiente ejemplo será más sencillo entenderlo:

```
class MiInteger {
    private int valor = 0;

    public MiInteger(int valor) {
        this.valor = valor
    }

    public int plus(MiInteger otro) {
        valor + otro.valor
    }
}

def mc1 = new MiInteger(8)
def mc2 = new MiInteger(7)
println mc1 + mc2
```

Tratamiento de todo como un objeto

Como todo programador *Java* sabe, este lenguaje distingue entre tipos primitivos (`int`, `double`, `float`...) y tipos de referencia (`Integer`, `String`, `Object`...). No es posible llamar a un método en un tipo primitivo. Las variables con tipos de referencia, en cambio, contienen una referencia a un objeto. Es por esto que en *Java* existe un *wrapper* para cada tipo primitivo, lo que significa que el tipo primitivo queda envuelto en un objeto. Por ejemplo, el *wrapper* para el tipo `int` es `Integer`.

En *Groovy*, al contrario que en *Java*, todo se considera un objeto, inclusive los tipos primitivos. *Groovy* siempre usa la clase *wrapper* en vez de la clase primitiva, y además permite utilizar las operaciones sobre los tipos básicos. La conversión de un tipo primitivo en un tipo *wrapper* se conoce como *boxing*, y la acción inversa es el *unboxing*. *Groovy* realiza estas operaciones siempre que es necesario. A este concepto se le llama *autoboxing*.

Ejemplo:

```
println println 2.floatValue()
```

En el ejemplo anterior, *Groovy* permite llamar al método `intValue()` de la clase `Integer` a pesar de que se esté invocando desde lo que a priori es un tipo primitivo.

Referencias seguras

Normalmente, en otros lenguajes de programación tales como *Java*, si se llama a un método sobre un objeto que apunte a `null`, saltará una `NullPointerException`. Para evitar esta situación, en *Groovy* existe e operador `'?'`, que previene que se lance la excepción y en su lugar, devuelve `null`. Por ejemplo:

```
MiInteger mi
mi?.plus(new MiInteger(10))
```

La utilización del operador `'?'` antes de la invocación al método `plus()` con un objeto no inicializado, evitará que se lance `NullPointerExcepción` y devolverá `null`.

Booleanos

En *Java*, solamente el valor `true` puede ser considerado verdadero, y solamente el valor `false`, puede ser considerado falso. En *Groovy*, en cambio, este concepto varía. Por ejemplo, siempre que se evalúe un valor cero, un valor `null`, un `String` vacío, una colección vacía, un array sin elementos o un `StringBuilder` / `StringBuffer` vacío, se devolverá `false`. En caso contrario, se devolverá `true`.

Ejemplo:

```
def lista = []
// ...
if(lista) {
    println "ArrayList con elementos"
} else {
    println "ArrayList vacio"
}
```

Excepciones

En *Java*, siempre que un método arroja alguna excepción de tipo chequeada, el compilador obliga a que haya que manejar dicha excepción.

En *Groovy* todas las excepciones son de tipo no chequeadas. Esto hace posible que no sea obligatorio declararlas ni capturarlas, cosa que da libertad para manejarlas solamente cuando sea necesario hacerlo.

2.1.4. Ejecución de código Groovy

Existen varias maneras de ejecutar código *Groovy*. Puede compilarse a *bytecodes* igual que en *Java*, generando archivos `.class` que se pueden ejecutar directamente en la JVM. Por otro lado, este código se puede ejecutar mediante *scripts*, a través del comando *groovy shell*, o mediante el comando *groovy console*.

Scripts

Si se tiene la sentencia de código `println "¡Hola mundo!"` y se guarda en un fichero de texto con extensión `.groovy` y cuyo nombre sea `holaMundo.groovy`, la ejecución de un *script* compilará el archivo en *bytecode Java* y será ejecutado en memoria. El archivo será una clase con un método `main` en el que se insertará el contenido del *script* que esté fuera de cualquier método, más los métodos del *script* que serán insertados dentro de la propia clase.

Para ejecutar un *script* en *Groovy* se debe escribir en la consola de comandos:

```
>> groovy nombreDelFichero.groovy
```

En el caso del ejemplo habría que escribir:

```
>> groovy holaMundo.groovy
```

Groovy shell

Otra opción para ejecutar código en *Groovy* es a través de *groovy shell*. Para ello, desde la consola de comandos hay que escribir el comando:

```
>> groovysh
```

El *prompt* cambiará y se podrá escribir directamente el código *Groovy* que se desea ejecutar. Por ejemplo, una vez dentro de *groovy shell*, si se escribe la línea de código `println "Hola mundo"`, como resultado de la ejecución aparecerá `Hola mundo`:

```
groovy:000> println "Hola mundo"
Hola mundo
```

Este comando es adecuado para ejecutar pequeños *scripts*, ya que algunas características más complejas del lenguaje aún no están soportadas.

Groovy console

Otro modo de ejecutar código *Groovy* es mediante el comando *groovy console*. Esta opción es más potente que la *shell* y no tiene limitaciones con respecto a características complejas del lenguaje, además funciona en modo gráfico (ver figura 2.2). Permite guardar y cargar archivos, editar texto, etc. Para ejecutar la *groovy console* hay que lanzar el comando:

```
>> groovyconsole
```

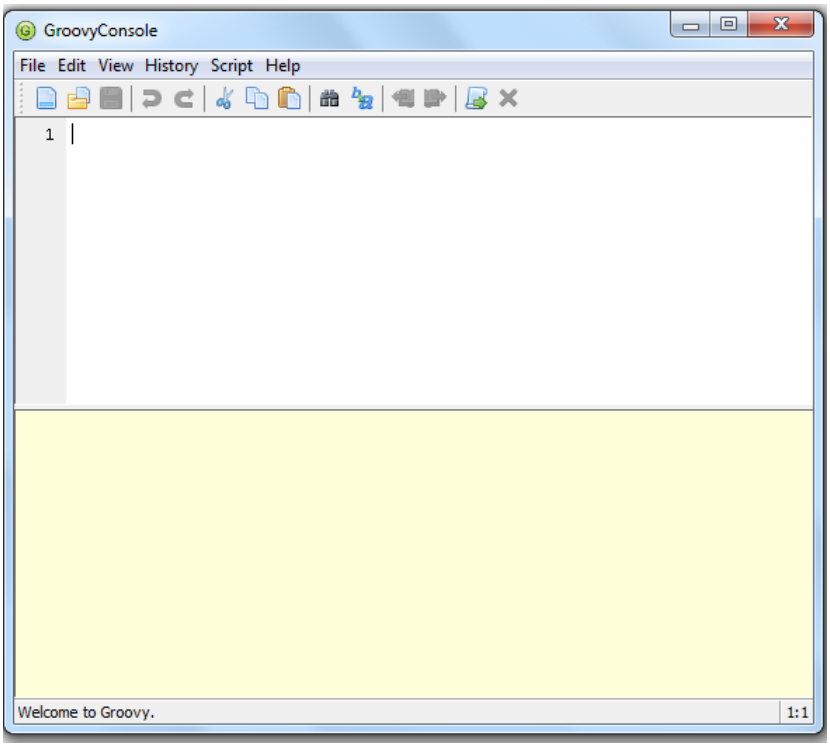


Figura 2.2: Groovy console

2.1.5. Cadenas

Groovy soporta tres tipos de cadenas de texto: `Strings`, `GStrings` y `Heredocs`.

Strings

El `String` en *Groovy* es similar a un `String` en *Java*, se diferencian únicamente en que se pueden usar tanto comillas simples como comillas dobles. Ejemplo:

```
def cadena1 = "Esta cadena es valida en Java y Groovy"
def cadena2 = 'Esta cadena es valida en Groovy'
```

Las comillas simples pueden contener comillas dobles en su interior sin necesidad de tener que escaparlas, y viceversa. Ejemplo:

```
def cadena3 = "Hola 'mundo'"
def cadena4 ='Hola "mundo"'
```

GStrings

Los `GStrings` son cadenas de texto que contienen en su interior expresiones. Solamente pueden construirse con comillas dobles (o triples), en este caso no son válidas las comillas simples. Ejemplo:

```
def edad = 25
println "Pepe tiene ${edad} primaveras"
```

En el ejemplo anterior, se observa que dentro de la cadena de texto se pasa una variable. Para indicar que se trata de una variable se utiliza el operador `${}`. La variable que se indica dentro del operador es evaluada en tiempo de ejecución, así que se puede indicar cualquier expresión dentro de un `GString`.

```
def edad = 25
def mensaje = "El dia ${new Date()} Pepe hace ${edad}
primaveras"
println mensaje
```

Heredocs

El tercer tipo de cadena de texto que soporta *Groovy*, es el `Heredoc`. Los `Heredocs` se forman con tres comillas simples o tres comillas dobles, y permiten almacenar cadenas de texto de varias líneas en una única variable (sin necesidad de escapar el salto de línea).

Los `Heredocs` permiten también mezclar comillas simples y comillas dobles en su interior sin necesidad de escaparlas. Ejemplo:

```
println """
linea uno
linea dos
linea tres con "comillas dobles" y 'comillas simples'
"""
println '''heredoc con comillas simples'''
println """heredoc con comillas dobles"""
println """heredoc con comillas dobles y expresion embebida:
${new Date()}"""
```

2.1.6. Expresiones regulares

En *Groovy*, las expresiones regulares se pueden manejar de un modo muy sencillo. Para ello existen tres operadores: el operador de búsqueda (`~`), el operador de coincidencia (`==~`), y el operador de patrón (`~cadena`).

Para delimitar cadenas en expresiones regulares, se pueden utilizar tanto comillas como barras. Ejemplos:

```
assert "abc" ==~ /abc/
assert "abcabc" ==~ /abc/ //Falla porque no es igual
assert "abc" ==~ /a.c/ //Empieza con a, un caracter, y c
assert "abc" ==~ /.*c/ //Cualquier cosa terminada en c
```

2.1.7. Listas, mapas y rangos

Java únicamente incluye nivel de soporte para los *arrays*, las demás colecciones se implementan como objetos estándar. En *Groovy*, en cambio, existe soporte sintáctico para listas y mapas. Además surge un nuevo tipo de lista, los rangos.

Ejemplos de rangos:

```
def a = 0..10
println a instanceof Range //devuelve true
println a.contains(6) //true
def d1 = new Date()
def d2 = d1 + 20
for (d in(d1..d2)) {
    println d
}
```

Ejemplos de listas:

```
def l1 = ['a', 'e', 'i','o']
def l2 = (1..100).toList()
// agregar elementos
l1 += 'u'
l2 << 101
// accede a los elementos
println l1
println l1[0..2]
l2.each{
    println it
}
```

Ejemplos de mapas:

```
def m = ['uno':1, 'dos':2, 'tres':3]
println m.uno //1
m.each {
    println "$it.key:$it.value"
}
```

2.1.8. Closures

Las *closures*, básicamente son bloques de código autónomos que se pueden pasar como parámetro a un método. Ejemplo:

```
def ejemploClosure = { println 'Hola mundo' }
```

En el ejemplo anterior se define un *closure*, siempre van entre llaves, y se asigna a una variable llamada `ejemploClosure`. Para ejecutar este *closure* se puede invocar como si fuera un método:

```
ejemploClosure()
```

Una *closure* también acepta parámetros:

```
def ejemploClosure2 = { println "Hola ${it}" }
ejemploClosure2 "mundo"
```

Como se puede observar en el ejemplo precedente, se utiliza la variable `it` para pasar parámetros. Esta variable es implícita para todas las *closures* que no definen parámetros. Si se quieren pasar parámetros explícitos:

```
def ejemploClosure3 = { nombre ->
    println "Hola ${nombre}"
}
ejemploClosure3 "mundo"
```

2.1.9. GroovyBeans

Groovy facilita la creación de *JavaBeans* ya que:

- Crea los *getters* y los *setters* automáticamente, sin necesidad de implementarlos.
- Simplifica la sintaxis de acceso a propiedades.
- Simplifica el registro de auditores a eventos.

Ejemplo:

```
class Libro {
    String titulo
    String autor
    int numPaginas
}
```

El ejemplo anterior define un libro con tres atributos que pueden ser leídos o escritos directamente. *Groovy* generará los *getters* y *setters* automáticamente en tiempo de compilación. Además, también se puede llamar a los atributos directamente por su nombre.

```
def libro = new Libro()
libro.titulo = "Los renglones torcidos de Dios"
libro.autor = "Torcuato Luca de Tena"
libro.numPaginas = 500
println libro.titulo
```

Los *getters* y *setters* se pueden sobreescribir en caso de que sea necesario modificar lo que se quiere devolver o almacenar.

2.1.10. Niveles de acceso

En *Groovy*:

- Todas las clases son públicas por defecto.
- Todos los métodos son públicos por defecto.
- Todos los atributos son públicos por defecto.

2.1.11. Constructores

Groovy ofrece una sintaxis especial para crear un objeto en una única línea:

```
def libro = new Libro(titulo:'Los renglones torcidos de Dios',
autor:'Torcuato Luca de Tena', numPaginas:500)
```

Los valores que se han pasado al constructor son en realidad un `Map`, así que también se puede pasar al constructor un `Map` que haya sido definido previamente. Ejemplo:

```
def mapa = [titulo: 'Los renglones torcidos de Dios',
autor:'Torcuato Luca de Tena', numPaginas:500]
def libro = new Libro(mapa)
println mapa.titulo
```

Además no es necesario pasar todos los atributos al crear un objeto. En el caso del ejemplo también sería válido:

```
def libro = new Libro(titulo:'Los renglones torcidos de Dios')
```

En caso de que se deseen proporcionar valores por defecto a los atributos que no se inicialicen en el constructor, se puede hacer. En la declaración del constructor los parámetros opcionales deben aparecer después de los parámetros obligatorios. Ejemplo:

```
class Libro {
    String titulo
    String autor
    int numPaginas

    Libro(int numPaginas, String titulo='Titulo por defecto',
    String autor='Autor por defecto') {
        this.titulo = titulo
        this.autor = autor
        this.numPaginas = numPaginas;
    }
}

def libro = new Libro(10)
```

2.2. Grails

2.2.1. Introducción

Grails[6][7][8] es un *framework* libre para aplicaciones web desarrollado sobre el lenguaje de programación *Groovy*. Su objetivo principal es lograr un desarrollo altamente productivo. Dicha productividad se consigue con los dos principios que el framework promulga: la convención sobre la configuración y el no repetir código (DRY, *Don't repeat yourself*). Además, *Grails* posee un entorno de desarrollo estandarizado que oculta al programador una gran parte de los detalles de configuración.

Grails, en un principio, fue conocido como *Groovy on Rails*. La primera versión nació en julio de 2005. A día de hoy, la última versión estable es la 2.0, que salió en diciembre de 2011.

2.2.2. Arquitectura

La arquitectura de una aplicación web implementada en *Grails* se basa en el patrón de diseño Modelo-Vista- Controlador (MVC).

En este tipo de arquitectura se pretende separar la lógica de la aplicación de los datos y de la interfaz de usuario.

- Modelo. Es la representación de la información con la que se opera en el sistema. La lógica de datos asegura la integridad de los mismos. En *Grails* el mapeo de los datos se realiza a través de GORM (*Grails Object-Relational Mapping*), que son clases escritas en *Groovy* que definen una entidad.
- Vista. Es la presentación del modelo, la interfaz con la que el usuario interactuará con el sistema. La vista la conforman un conjunto de páginas web que se crean a partir de ficheros fuentes de tipo GSP, las denominadas *Groovy Server Pages*. Los GSP mezclan etiquetas de HTML con otras nativas de *Grails*.
- Controlador. Es la parte que responde a los eventos que son provocados por el usuario a través de las vistas.

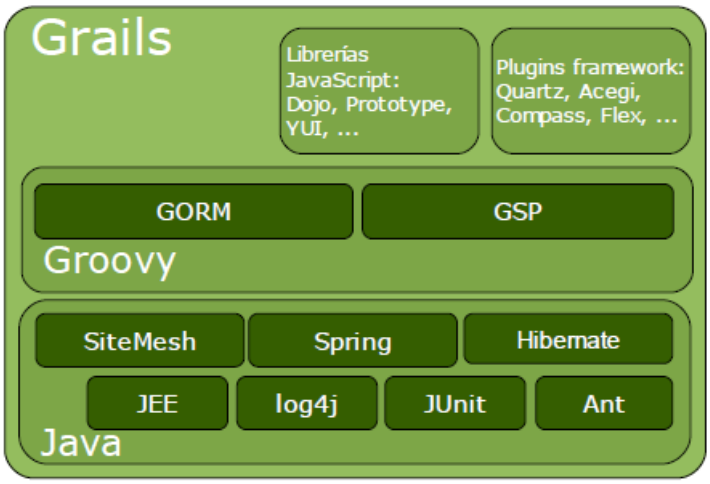


Figura 2.3: Arquitectura Grails

Es necesario precisar que *Grails* se sustenta en varios pilares fundamentales (ver figura 2.3):

- *Groovy*, para la creación de propiedades y objetos dinámicos en las entidades de la aplicación.
- *Spring*, para los flujos de trabajo y la inyección de dependencias entre componentes.
- *Hibernate*, para la persistencia de los datos.
- *SiteMesh*, para la composición de las vistas.
- *Ant*, para la gestión del proceso de desarrollo.

2.2.3. Instalación

Para instalar *Grails* hay que descargar el fichero `.zip` disponible en la página `http://grails.org/Download` con la última versión.

Suponiendo que se va a instalar en un entorno *Windows*:

- Descomprimir el fichero en la ruta en la que se quiera instalar *Grails*, por ejemplo, `C:\Program Files\grails-2.0.1`.
- Crear la variable de entorno GRAILS_HOME apuntando a la ruta en la que se ha realizado la extracción del archivo.
- Si no está ya, crear la variable de entorno JAVA_HOME apuntando a la ruta dónde esté instalado el JDK de *Java*.
- Por último, añadir una referencia al directorio `C:\Program Files\grails-2.0.1\bin` a la variable de entorno PATH. Nótese que tanto la variable de entorno PATH como GRAILS_HOME deben estar definidas en el mismo nivel de variables de entorno (o ambas en las variables de usuario, o ambas en las variables del sistema).

2.2.4. Características

- Ofrece un *framework* web altamente productivo para la plataforma *Java*.
- Reutiliza otros *framework Java* tales como *Spring* e *Hibernate*.
- Ofrece un *framework* que sea fácil de aprender y utilizar para el programador.
- Dota al usuario de un *framework* de persistencia potente y consistente.
- Dota al usuario de patrones de visualización como los GSP y de bibliotecas de etiquetas dinámicas para crear componentes web de manera más sencilla.
- Ofrece un buen soporte de AJAX, sencillo de utilizar y personalizable.
- Ofrece un entorno de desarrollo orientado a pruebas.
- Ofrece un entorno de desarrollo muy completo que incluye servidor web y recarga automática de recursos.

Para incrementar la productividad y minimizar el tiempo de desarrollo de aplicaciones web, *Grails*:

- Elimina la tarea inicial de configuración del *framework* mediante ficheros XML. En *Spring* es necesaria este tipo de configuración y ello implica que la productividad de los programadores disminuye, ya que tienen que gastar mucho tiempo configurando y manteniendo los *frameworks* mientras la aplicación crece. En *Grails* es posible la configuración directa a través del código de la aplicación. Por ejemplo, un controlador web es una clase cuyo nombre termina con la palabra clave `Controller`; y una clase de dominio, es una clase cuyo nombre termina por la palabra clave `Domain`.
- Posee un servidor web integrado. Usando herramientas *Java* tradicionales, es tarea del desarrollador ensamblar los componentes. *Grails*, en cambio, tiene integrado un servidor web preparado para desplegar la aplicación desde un primer momento. Todas las librerías necesarias son parte de la distribución, y están preparadas para ser desplegadas automáticamente.
- Posee métodos dinámicos en algunas de sus clases. Estos métodos se añaden a la clase en tiempo de ejecución como si su funcionalidad se hubiese compilado, y permiten a los programadores realizar operaciones sin tener que hacer implementación de interfaces o herencia de otras clases. Un ejemplo de clase con métodos dinámicos son las clases de dominio, que contienen métodos para automatizar operaciones tales como salvar (*save*), borrar (*delete*) y buscar (*find*).

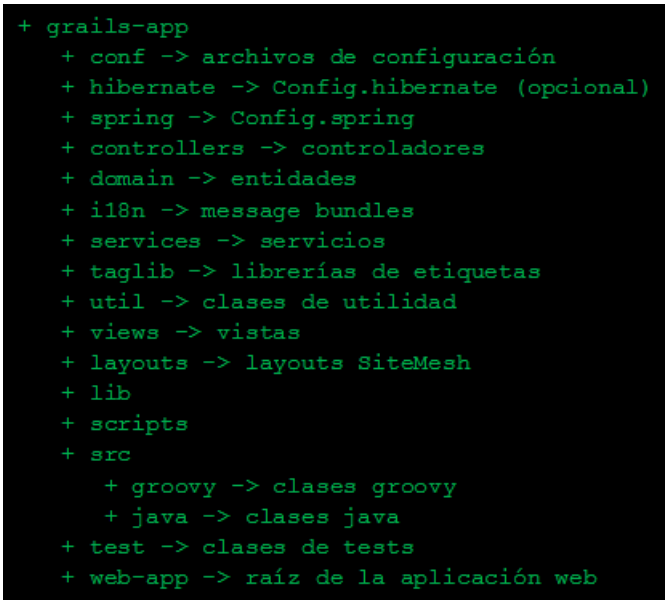


Figura 2.4: Estructura aplicación Grails

2.2.5. Formato de una aplicación Grails

Grails contiene todo lo necesario para desarrollar, no es necesario un servidor adicional, ni siquiera base de datos. Aún así, siempre se pueden usar otros.

Para comenzar una aplicación en *Grails* hay que situarse en el directorio en el que se desea crear el proyecto e introducir el comando:

```
>> grails create-app nombreProyecto
```

Se creará una carpeta con el nombre del proyecto que se haya indicado y con la estructura que se muestra en la figura 2.4.

2.2.6. Controladores

En *Grails*, los controladores se usan para recibir las instrucciones del usuario a través de las vistas y aplicar la lógica de negocio sobre el modelo, decidiendo así que vista se debe mostrar.

Una clase que haga las funciones de controlador, por convención, debe finalizar con la palabra `Controller` (`XXXController.groovy`). A continuación se muestra un ejemplo sencillo de controlador:

```
class BookController {
    def list = {
        [ books: Book.findAll() ]
    }
 }
```

El controlador anterior tiene implementada una *closure* denominada `list` que accede al listado de libros dados de alta en la base de datos y los devuelve.

Para crear un controlador en *Grails* se utiliza el siguiente comando:

```
>> grails create-controller nombreDelControlador
```

En el caso del ejemplo, habría que ejecutar:

```
>> grails create-controller book
```

Dicho comando recoge el nombre del controlador y crea una clase en el directorio `grails-app/controllers` del proyecto *Grails*. Al ser un controlador, *Grails* lo reconoce como tal y mapea la acción `list` a la dirección del entorno de desarrollo `http://localhost:8080/book/list`.

2.2.7. Servicios

Muchos desarrolladores implementan la lógica de negocio directamente en los controladores, pero esto es una práctica desaconsejable y poco ortodoxa. Lo mejor es incluir otra capa, usada por la de control, en la que se implemente toda la lógica de la aplicación. De este modo habrá una mayor claridad en el código.

Un servicio es una clase que termina siempre con la palabra clave `Service` (`XXXService.groovy`) y para crearlo hay que emplear el comando:

```
>> grails create-service nombreDelServicio
```

El servicio creado se alojará en el directorio `grails-app/services` del proyecto *Grails*. Para usar un servicio desde otros componentes tales como un controlador, se puede declarar una variable cuyo nombre sea el servicio en cuestión. De este modo se activará la inyección automática de la dependencia por parte de *Spring*.

Ejemplo de un servicio y su controlador:

```
class BookService {
    void cutPriceOfBook(Book book) {
        // implementar logica del servicio
    }
}
```

```
class BookController {
    def bookService
       def cutPrice = {
       def b = Book.get(params.id)
       bookService.cutPriceOfBook(b)
    }
    def list = {
       [ books: Book.findAll() ]
    }
}
```

2.2.8. Vistas

Para generar las vistas *Grails* soporta JSP y GSP. Los GSP son una versión simplificada de los JSP utilizados en *Java*. Permiten intercalar en el código etiquetas HTML y expresiones, además de etiquetas propias de las GSP.

A continuación se muestra una página GSP asociada al ejemplo anterior del `BookController`:

```
<html>
  <head>
    <title>Our books</title>
  </head>
  <body>
    <ul>
      <g:each in="\${books}">
        <li>\${it.title} (\${it.author.name})</li>
      </g:each>
    </ul>
  </body>
</html>
```

La vista del ejemplo debe encontrarse en el directorio `grails-app/views/book/list.gsp` del proyecto *Grails*. Esta localización se mapea automáticamente por el `BookController` y la acción `list`.

Por último, si se desea invocar a una vista desde un controlador:

```
class ExampleController {
    def  actionExample = {
       render (view:'list')
    }
}
```

2.2.9. Ayudas para AJAX

Grails soporta varias librerías AJAX, incluidas *OpenRico*, *Prototype*, *Dojo*, etc. Existen librerías de *tags* para crear HTML con código *Javascript* en las páginas y también para hacer llamadas AJAX al servidor.

2.2.10. Librerías de tags

Grails también proporciona un gran número de librerías de *tags*. Además, se pueden crear librerías nuevas y reusarlas.

En *Grails* una librería de etiquetas es una clase *Groovy* cuyo nombre tiene el formato `XXXTagLib.groovy`, y que se aloja en el directorio `grails-app/taglib`. Por ejemplo:

```
def formatDate = { attrs ->
    out << new java.text.SimpleDateFormat(attrs.format).format(attrs.date)
}
```

El *tag* `formatDate` formatea el objeto `java.util.Date` a `String`.

A continuación se muestra un ejemplo de uso del *tag* creado en un GSP:

```
<g:formatDate format="yyyyMMdd" date="\${myDate}"/>
```

2.2.11. Clases de dominio

En *Grails* para persistir datos y mapearlos, son necesarias las clases de dominio.

Definición del modelo de datos

Los datos modelados en las clases de dominio, se almacenan en la base de datos mediante GORM. Una clase de dominio se crea mediante el siguiente comando:

```
>> grails create-domain-class nombreDelDatoAModelar
```

A continuación se pone como ejemplo una clase de dominio que modela un libro:

```
class Book {
     String title
     Person author
}
```

Al crear esta clase, ya está modelada para poder ser persistida de manera automática por *Grails*. A partir de la versión 0.3 de *Grails*, GORM ha sido mejorado y añade las propiedades `id` y `version` automáticamente si no se declaran. En la propiedad `id` se almacena la clave primaria de la tabla (un identificador autoincremental). La propiedad `version` se utiliza para gestionar el bloqueo optimista.

A partir de estas clases también se pueden modelar las relaciones entre distintas entidades (1:1, 1:N, N:1 y N:M).

Validaciones

GORM permite restringir los valores que se pueden asignar a las distintas propiedades de una entidad. Esto se realiza a través de una propiedad estática denominada `contraints`.

Esta propiedad es muy importante puesto que no solo se emplea para restringir valores, sino también para crear el esquema de datos o para la generación de vistas mediante *scaffolding*.

Es una *closure* en la que se definen, una por una, las reglas de validación de cada una de las propiedades que conforman una entidad. Para aplicar estas restricciones, *Grails* utiliza la librería *Apache Commons Validator*. Algunas de las reglas que se pueden definir son:

- `blank`. Habrá que poner `blank:false` si la propiedad no admite cadenas de texto vacías.
- `nullable`. Por defecto *Grails* no permite valores nulos en las propiedades de una entidad. Si se desea que una propiedad pueda contener un valor nulo, entonces habrá que indicar `nullable:true`.
- `creditCard`. Indica que un campo debe contener valores válidos para números de tarjetas de crédito.
- `email`. Especifica que un campo debe contener valores válidos para una dirección de correo electrónico.

- `inList`. Indica que el valor de un campo debe estar entre los de una lista cerrada dada. Por ejemplo: `animal(inList:['perro', 'gato'])`.
- `matches`. Obliga a que el valor del campo valide una expresión regular dada.
- `max`. Obliga a que el valor del campo no sea mayor que el del límite indicado.
- `maxSize`. Obliga a que el campo no tenga un tamaño que sobrepase el límite indicado.
- `min`. Obliga a que el valor del campo no sea menor que el del límite indicado.
- `minSize`. Obliga a que el tamaño del campo no tenga un valor inferior que el del límite indicado.
- `notEqual`. Obliga a que el valor de la propiedad no pueda ser igual a uno indicado.
- `range`. Para restringir los valores de una propiedad en un rango determinado.
- `scale`. Indica el número de decimales para los campos que sean valores en coma flotante.
- `size`. Restringe el tamaño mínimo y máximo a la vez.
- `unique`. Garantiza que los valores de la propiedad sean únicos.
- `url`. Obliga a que el valor de la propiedad sea una cadena que represente una URL válida.
- `validator`. Permite definir validaciones especiales para casos que no estén cubiertos en las validaciones que se han indicado anteriormente.

Mensajes de error en las validaciones

Al aplicarse las validaciones sobre los campos de la entidad en *Grails*, si éstas no validan se lanza un mensaje de error. Para ello, *Grails* utiliza los archivos de recursos que contiene la carpeta `grails-app/i18n`. De este modo la aplicación es compatible con distintos idiomas.

Si se desea modificar estos mensajes que vienen por defecto, se puede hacer mediante la manipulación de estos archivos, añadiendo el mensaje personalizado que se desea que aparezca para una propiedad en cuestión.

En general, *Grails* buscará claves con el siguiente formato:

```
[Clase].[Propiedad].[Restricción]
```

Por ejemplo, si se desea modificar el mensaje que aparece cuando no se inserta un título para la propiedad homónima de una clase denominada `Libro`, entonces habrá que modificar el mensaje asociado a la clave:

```
Libro.titulo.blank
```

Si no se encuentra ningún mensaje con ese formato, *Grails* mostrará el mensaje por defecto asociado a la clave:

```
default.invalid.[Restricción].mensaje
```

Métodos de las clases de dominio

Las clases de dominio poseen distintos métodos dinámicos y estáticos que posibilitan el acceso a los objetos almacenados en la base de datos, así como su modificación, su borrado, etc.

En cuanto a los métodos dinámicos de instancia están:

- `save()`, que guarda un objeto en la base de datos.

```
def book = new Book(title:"Farenheit 451",
author:Author.findByName("Ray Bradbury"))
book.save()
```

- `delete()`, que borra un objeto de la base de datos.

```
def book = Book.findByTitle("Farenheit 451")
 book.delete()
```

- `refresh()`, que refresca el estado de un objeto de la base de datos.

```
def book = Book.findByTitle("Farenheit 451")
book.refresh()
```

- `ident()`, que recupera el identificador del objeto de la base de datos.

```
def book = Book.findByTitle("Farenheit 451")
def id = book.ident()
```

Por otro lado, también existen métodos dinámicos de clase, que son:

- `count()`, que recupera el número de registros de una clase dada.

```
def bookCount = Book.count()
```

- `exists()`, que devuelve verdadero si el objeto con el identificador dado existe en la base de datos.

```
def bookExists = Book.exists(1)
```

- `find()`, que devuelve el primer objeto de la base de datos que cumple con la consulta realizada, pasada como parámetro en lenguaje HQL *Hibernate*.

```
 def book = Book.find("from Book b where b.title = ?",
 [ 'Farenheit 451' ])
```

- `findAll()`, que devuelve todos los objetos existentes en la base de datos. También se puede pasar como parámetro una consulta HQL.

```
def books = Book.findAll()
def books2 = Book.findAll("from Book")
```

- `findBy*()`, que devuelve el primer objeto de la base de datos que coincide con el patrón indicado. El * debe sustituirse por alguno de los atributos de la entidad de la que se realiza la consulta.

```
def book = Book.findByTitle("Farenheit 451")
def book2 = Book.findByTitleLike("%451%")
```

- `findAllBy*()`, que devuelve una lista de objetos de la base de datos que coincide con el patrón indicado. El * indica lo mismo que en el caso anterior.

```
def books = Book.findAllByTitleLike("Far%")
```

- `findWhere*()`, que devuelve el primer objeto de la base de datos que coincide con los parámetros indicados.

```
def book = Book.findWhere(title:"Farenheit 451")
```

2.2.12. Scaffolding

Grails soporta un mecanismo muy potente denominado *scaffolding*. Una vez creado el modelo de datos, se puede solicitar a *Grails* que genere el controlador y las vistas necesarias para realizar operaciones CRUD (*Create*, *Read*, *Update* y *Delete*) con las entidades modeladas en las clases de dominio. Siguiendo con el ejemplo de la clase de dominio `Book`, si se desea generar una interfaz web que permita realizar operaciones CRUD, bastaría con crear un controlador como el siguiente:

```
class BookController {
     def scaffold = true
}
```

Creando esta clase se pueden realizar operaciones CRUD a través de la dirección `http://localhost:8080/nombreAplicación`.

En las imágenes siguientes 2.5, 2.6, 2.7, se puede observar como quedaría la interfaz web de la clase de dominio `Book` generada mediante esta técnica.

El *scaffolding* es recomendable únicamente en casos en los que la interfaz web que se requiere es sencilla, para hacer operaciones CRUD, si se desea un modelo más personalizado lo mejor es hacer uso de los GSP y el HTML.

2.2.13. Ficheros de configuración

En *Grails* hay una serie de ficheros que se incluyen en el directorio `grails-app/conf`. Dichos ficheros se utilizan para configurar la aplicación. Los más importantes se explican a continuación.

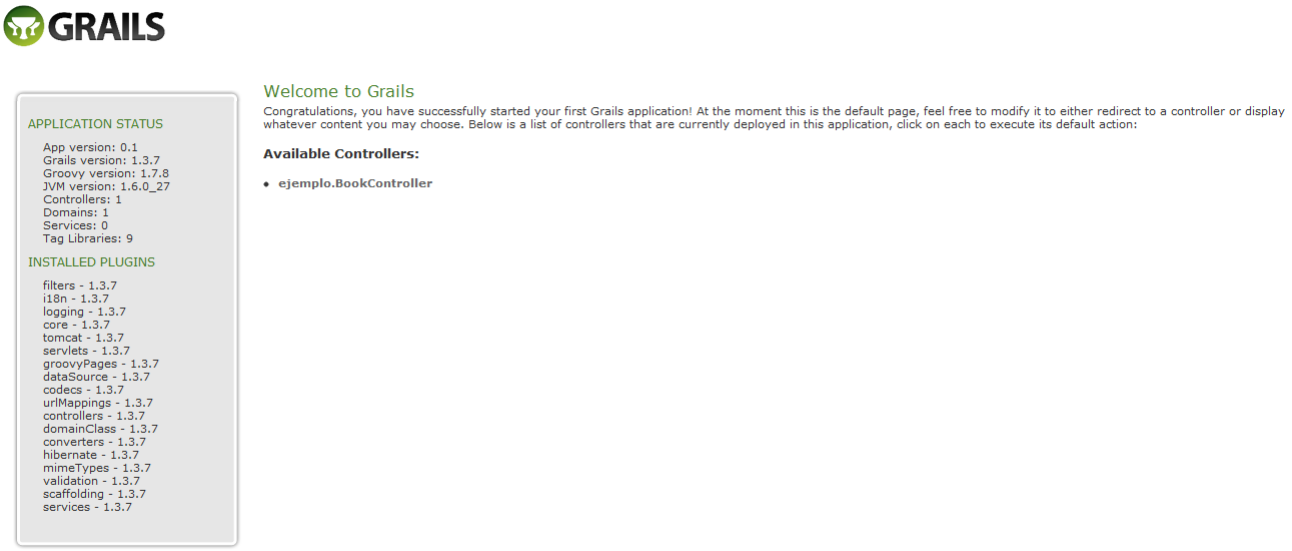


Figura 2.5: Scaffolding: Vista principal de la aplicación de ejemplo

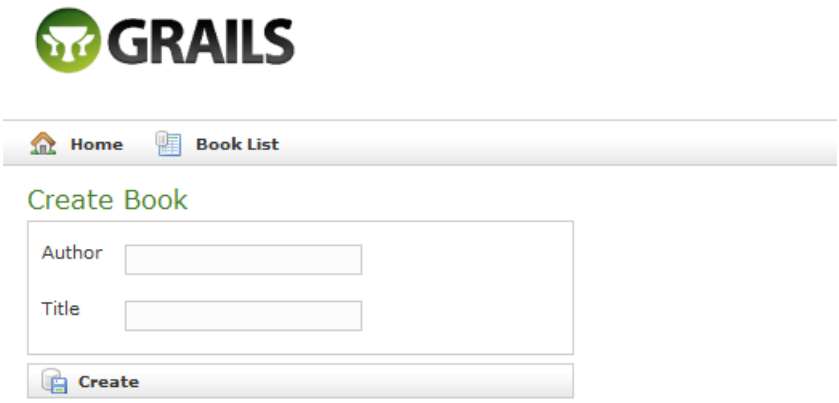


Figura 2.6: Scaffolding: CREATE

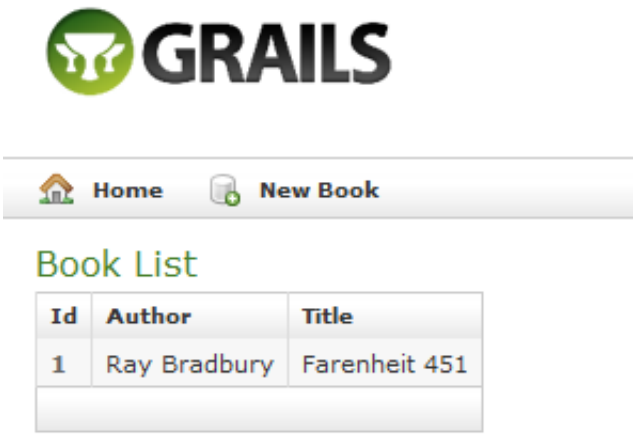


Figura 2.7: Scaffolding: READ

Config.groovy

Este archivo contiene los parámetros de configuración generales de la aplicación. Permite la declaración de variables y el uso de tipo de datos, y además, cualquier parámetro que se defina en este archivo estará disponible desde cualquier artefacto de la aplicación (controlador, servicio...) a través del objeto global `grailsApplication.config`.

Ejemplo de fichero `Config.groovy`:

```
// change this to alter the default package name
// and Maven publishing destination
grails.project.groupId = appName
// enables the parsing of file extensions from
// URLs into the request format
grails.mime.file.extensions = true
grails.mime.use.accept.header = false
grails.mime.types = [
                  html: ['text/html','application/xhtml+xml'],
                  xml: ['text/xml', 'application/xml'],
                  text: 'text/plain',
                  js: 'text/javascript',
                  rss: 'application/rss+xml',
                  atom: 'application/atom+xml',
                  css: 'text/css',
                  csv: 'text/csv',
                  all: '*/*',
                  json: ['application/json','text/json'],
                  form: 'application/x-www-form-urlencoded',
                  multipartForm: 'multipart/form-data'
                  ]

// URL Mapping Cache Max Size, defaults to 5000
//grails.urlmapping.cache.maxsize = 1000

// What URL patterns should be processed by the
// resources plugin
```

```
grails.resources.adhoc.patterns =
['/images/*', '/css/*', '/js/*', '/plugins/*']


// The default codec used to encode data with ${}
grails.views.default.codec = "none"
// none, html, base64
grails.views.gsp.encoding = "UTF-8"
grails.converters.encoding = "UTF-8"
// enable Sitemesh preprocessing of GSP pages
grails.views.gsp.sitemesh.preprocess = true
// scaffolding templates configuration
grails.scaffolding.templates.domainSuffix = 'Instance'

// Set to false to use the new Grails 1.2
// JSONBuilder in the render method
grails.json.legacy.builder = false
// enabled native2ascii conversion of i18n
//properties files
grails.enable.native2ascii = true
// packages to include in Spring bean scanning
grails.spring.bean.packages = []
// whether to disable processing of multi part requests
grails.web.disable.multipart=false

// request parameters to mask when logging exceptions
grails.exceptionresolver.params.exclude = ['password']

// enable query caching by default
grails.hibernate.cache.queries = true

// set per-environment serverURL
//stem for creating absolute links
environments {
    development {
        grails.logging.jul.usebridge = true
    }
    production {
        grails.logging.jul.usebridge = false
        // TODO: grails.serverURL = "http://www.changeme.com"
    }
}

// log4j configuration
log4j = {
    // Example of changing the log pattern for
    // the default console
    // appender:
    //
    //appenders {
    //    console name:'stdout',
          layout:pattern(conversionPattern: '%c{2} %m%n')
    //}

    error  'org.codehaus.groovy.grails.web.servlet',
    // controllers
```

```
            'org.codehaus.groovy.grails.web.pages',
            // GSP
            'org.codehaus.groovy.grails.web.sitemesh',
            // layouts
            'org.codehaus.groovy.grails.web.mapping.filter',
            mapping
            'org.codehaus.groovy.grails.commons',
            // core / classloading
            'org.codehaus.groovy.grails.plugins',
            // plugins
            'org.codehaus.groovy.grails.orm.hibernate',
            // hibernate integration
            'org.springframework',
            'org.hibernate',
            'net.sf.ehcache.hibernate'
}
```

Como se puede observar a través de este fichero, se puede configurar la generación de trazas mediante `log4j`.

DataSource.groovy

Como *Grails* se basa en *Java*, la configuración de acceso a datos recae en JDBC. El archivo `DataSource.groovy` contiene los parámetros de configuración de la base de datos en cada uno de los entornos de desarrollo. En el primer bloque, `dataSource`, se deben definir los parámetros genéricos de configuración para la base de datos, que podrán ser sobreescritos en cada uno de los entornos.

Ejemplo:

```
dataSource {
        pooled = true
        driverClassName = "com.mysql.jdbc.Driver"
        dialect = "org.hibernate.dialect.MySQL5InnoDBDialect"
        username = "root"
        password = "root"
        dbCreate = "update"
}
hibernate {
        cache.use_second_level_cache = true
        cache.use_query_cache = true
        cache.provider_class =
        'net.sf.ehcache.hibernate.EhCacheProvider'
}
// environment specific settings
environments {
        development {
                dataSource {
                        url = "jdbc:mysql://localhost:3306/bd"
                }
        }
        test {
                dataSource {
                        url = "jdbc:mysql://localhost:3306/bd"
                }
```

```
        }
        production {
                dataSource {
                        url = "jdbc:mysql://localhost:3306/bd"
                }
        }
}
```

UrlMappings.groovy

Grails, por defecto, utiliza el siguiente convenio para formar las URLs empleadas en una aplicación web.

`/controlador/acción/id`

Si se desea personalizar estas URLs, esto se puede hacer mediante el archivo `UrlMappings.groovy`.

Ejemplo:

```
class UrlMappings {
        static mappings = {
                "/chapters"{
                        controller = "book"
                        action = "list"
                }
        }
}
```

En el caso del ejemplo, se indica que cuando el usuario acceda a la URL `/chapters`, debería ejecutarse la acción `list` del controlador `book`.

2.3. Quartz

2.3.1. Introducción

Muchas aplicaciones requieren programar tareas que se ejecuten de manera automática, tales como: enviar todos los viernes un correo electrónico a los clientes de una empresa, comprobar si se han recibido nuevos pedidos de clientes diariamente, etc.

Crear un sistema que soporte tareas automáticas es muy sencillo si se utilizan *frameworks* ya disponibles. En concreto, *Quartz*[9][10][11] es un *framework* de código abierto con licencia *Apache 2.0* muy aconsejable para planificar y gestionar este tipo de tareas.

2.3.2. Características

- Es válido tanto para aplicaciones JEE como para JSE.
- Posee planificación flexible de tareas mediante expresiones *cron* o *triggers*.
- Mantiene el estado de las tareas incluso en caso de fallos o reinicios de la máquina.
- Hay posibilidad de participar en transacciones JTA.

- Hay posibilidad de trabajar en modo clúster.
- Posee una API muy completa, con multitud de clases y varios tipos de escuchadores de eventos (`JobListener`, `TriggerListener` y `SchedulerListener`)

En los múltiples *plugins* existentes para *Grails*, hay uno implementado que está preparado para utilizar *Quartz*.

Este *plugin* permite programar *jobs* para que éstos sean ejecutados usando intervalos especificados de tiempo o expresiones *cron*.

2.3.3. Instalación del plugin Quartz

Desde el directorio principal de la aplicación hay que ejecutar la siguiente instrucción:

```
>> grails install-plugin quartz
```

2.3.4. Uso

Programando jobs con triggers

Para crear un nuevo *job* hay que ejecutar el siguiente comando:

```
>> grails create-job MyJob
```

Grails creará un nuevo *job* y lo colocará en un nuevo directorio, `grails-app/jobs`.

```
class MyJob {
  static triggers = {
    simple name: 'mySimpleTrigger', startDelay: 60000
    , repeatInterval: 1000
  }
  def group = "MyGroup"
  def execute(){
    print "Job run!"
  }
}
```

En el ejemplo de código anterior se esperará un minuto y después se llamará al método `execute` cada segundo. El intervalo de repetición (`repeatInterval`) y la demora inicial (`startDelay`) vienen dados en milisegundos y deben ser de tipo `Integer` o `Long`. Si estas propiedades no se especifican, los valores por defecto son un minuto para el intervalo de repetición y treinta segundos para la demora inicial. Los *jobs*, de manera opcional, pueden colocarse en diferentes grupos.

Los nombres de cada uno de los *triggers* que se definan deben ser únicos en la aplicación. Es importante también indicar que, por defecto, los *jobs* no se inician cuando se está ejecutando la aplicación en el entorno de *test*, aunque esto se puede modificar en su archivo de configuración.

Programando jobs con cron

Los *jobs* pueden programarse utilizando expresiones *cron*. Por ejemplo:

```
class MyJob  {
  static triggers = {
    cron name: 'myTrigger', cronExpression: "0 00 12 ? * *"
  }
 def group = "MyGroup"
 def execute(){
   print "Job run!"
 }
}
```

La expresión *cron* del código anterior se ejecutará diaramente, en concreto a las doce de la tarde.

Lo campos en una expresión *cron* son siete, y el formato es el que sigue:

`cronExpression: "s m h D M W Y"`

- `s`: segundos, acepta números enteros del 0 al 59 y los caracteres especiales `','`, `'-'`, `'*'` y `'/'`.
- `m`: minutos, acepta números enteros del 0 al 59 y los caracteres especiales `','`, `'-'`, `'*'` y `'/'`.
- `h`: horas, acepta números enteros del 0 al 23 y los caracteres especiales `','`, `'-'`, `'*'` y `'/'`.
- `D`: dias del mes, acepta números enteros del 1 al 31 y los caracteres especiales `','`, `'-'`, `'*'`, `'?'`, `'/'`, `'L'` y `'W'`.
- `M`: meses, acepta números enteros del 1 al 12 o las cadenas JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV y DEC, y los caracteres especiales `','`, `'-'`, `'*'` y `'/'`.
- `W`: días de la semana, acepta números enteros del 1 al 7 o las cadenas SUN, MON, TUE, WED, THU, FRI y SAT, y los caracteres especiales `','`, `'-'`, `'*'`, `'?'`, `'/'`, `'L'` y `'#'`.
- `Y`: año, es opcional. Acepta los valores de 1970 a 2099 y los caracteres especiales `','`, `'-'`, `'*'` y `'/'`.

Hay que notar que:

- El año es el único campo opcional y puede ser omitido. El resto de los campos son obligatorios.
- El día de la semana y el mes no son susceptibles a mayúscula y a minúscula. Es decir, Dec = DEC = dec.
- El día de la semana o el día del mes deben ser `'?'`. No se pueden especificar los dos valores simultáneamente. Tampoco se pueden marcar ambos con el caracter `'*'`, que es el comodín.

A continuación se muestran algunos ejemplos de expresiones *cron*:

- `0 0 12 * * ?`. Se ejecuta a las 12:00 PM cada día.
- `0 15 10 ? * *`. Se ejecuta a las 10:15 AM cada día.
- `0 15 10 * * ? 2005`. Se ejecuta a las 10:15 AM cada día durante el año 2005.

- `0 0/5 14 * * ?`. Se ejecuta cada 5 minutos, comenzando a las 2:00 PM y terminando a las 2:55 PM cada día.
- `0 15 10 ? * MON-FRI`. Se ejecuta a las 10:15 AM de lunes a viernes.

2.3.5. Múltiples triggers para un job

También se pueden registrar múltiples *triggers* para un mismo *job*, tal y como se muestra en el siguiente fragmento de código:

```
class MyJob {
    static triggers = {
        simple name: 'simpleTrigger', startDelay:10000,
        repeatInterval: 30000,
        repeatCount: 10
        cron name: 'cronTrigger', startDelay:10000,
        cronExpression: '0/6 * 15 * * ?'
        custom name: 'customTrigger',
        triggerClass: MyTriggerClass,
        myParam:myValue, myAnotherParam:myAnotherValue
    }
    def execute() {
        println "Job run!"
    }
}
```

Se pueden definir tres tipos de *triggers* distintos, que son:

1. `simple`, cuyos parámetros son:
   - `name`, que es el nombre que identifica de manera única al *trigger*.
   - `startDelay`, que es la demora inicial hasta que se empieza a ejecutar el *job*. Viene dada en milisegundos.
   - `repeatInterval`, es el *timeout* dado en milisegundos entre consecutivas repeticiones de un *job*.
   - `repeatCount`, es el *trigger* que se disparará *repeatCount* + 1 veces y se detendrá después. Si se indica 0, habrá una sola repetición. Si se indica -1, las repeticiones serán indefinidas.
2. `cron`, con parámetros:
   - `name`, que es el nombre que identifica de manera única al *trigger*.
   - `startDelay`, que es la demora inicial hasta que se empieza a ejecutar el *job*. Viene dada en milisegundos.
   - `cronExpression`, la expresión *cron*.
3. `custom`, cuyos parámetros son:
   - `triggerClass`, clase personalizada que implementa la interfaz *trigger*.
   - parámetros necesitados por el *trigger*.

Jobs dinámicos

También se pueden ejecutar *jobs* de manera dinámica. Los métodos disponibles para este cometido son:

- `MyJob.schedule(String cronExpression, Map params?)`, crea un *trigger* mediante una expresión *cron*.
- `MyJob.schedule(Long repeatInterval, Integer repeatCount?, Map params?)`, crea un *trigger* de tipo *simple*.
- `MyJob.schedule(Date scheduleDate, Map params?)`, programa un *job* en la fecha especificada.
- `MyJob.schedule(Trigger trigger)`, programa la ejecución de *jobs* con *triggers* personalizados.
- `MyJob.triggerNow(Map params?)`, fuerza la ejecución inmediata de un *job*.

Cada método, excepto el del *trigger* personalizado, tiene como opcional el parámetro *params*, que se puede utilizar para pasar datos al *job*.

```
class MyJob {
  def execute(context) {
    println context.mergedJobDataMap.get('foo')
  }
}
// now in your controller (or service, or something else):

MyJob.triggerNow([foo:"It Works!"])
```

Usando JobExecutionContext

Se puede definir el método `execute` de un *job* como `def execute(context)`, al que se le pasa como parámetro un `JobExecutionContext` de *Quartz*. Desde este contexto se puede obtener información del *job* tal como el nombre del *trigger* que se dispara, el tiempo de ejecución previo, el tiempo en el que se lanzará la siguiente ejecución, etc. Los parámetros del *job* están disponibles a través de `context.getMergedJobDataMap()`. Si el *job* se desea que sea con estado, sus datos también se persistirán por cada ejecución del *job*.

Configurando el plugin

Tras instalar el plugin de *Quartz*, se puede generar un fichero de configuración introduciendo el siguiente comando:

```
>> grails install-quartz-config
```

Esto generará un fichero de configuración denominado `QuartzConfig.groovy`, que se sitúa en el directorio `grails-app/conf`. Inicialmente el fichero tiene la siguiente configuración por defecto:

```
quartz {
    autoStartup = true
    jdbcStore = false
}
```

```
environments {
    test {
        quartz {
            autoStartup = false
        }
    }
}
```

Actualmente soporta las siguientes opciones de configuración:

- `autoStartup`: controla el comienzo automático del planificador de *Quartz* durante el inicio de la aplicación (por defecto es `true`).

- `jdbcStore`: establece si se desea persistir los *jobs* en la base de datos, por defecto es `false`. Para ello se necesitará proveer un fichero de propiedades denominado `quartz.properties`, así como crear las tablas pertinentes en la base de datos en la que se vayan a almacenar los *jobs*.

Logging

Los *logs* son inyectados dentro de la clase del *job* automáticamente. Para establecer el nivel del *logging*, hay que añadir una línea como la que sigue en el fichero `grails-app/conf/Config.groovy`:

```
debug 'grails.app.jobs'
```

Configuración de ejecución concurrente

Por defecto los *jobs* son ejecutados de manera concurrente, de modo que un nuevo *job* puede iniciarse si una ejecución previa del mismo *job* está aún en proceso. Si se desea sobreescribir este comportamiento se puede usar la propiedad `concurrent`.

```
concurrent = false
```

2.4. API REST de Twitter

Desde la API REST de *Twitter*[12] se puede obtener toda la información a la que un usuario puede acceder a través de la interfaz web de `twitter.com`. Para acceder a muchos de estos recursos REST se requiere autenticación, y el formato en el que se obtienen los datos puede ser XML, ATOM, JSON o RSS. Esto significa que si, por ejemplo, un usuario quiere acceder a sus mensajes directos a través de la API REST, éste necesitará estar autenticado. La autenticación utilizada en estos casos es *OAuth*, un protocolo que se explicará detalladamente en siguientes apartados de esta sección.

Por otra parte, es importante indicar que *Twitter* limita el acceso a la API por número de peticiones realizadas. En el caso de la API REST, la limitación es de 350 peticiones/hora siempre y cuando el usuario esté autenticado al realizar la petición a un recurso REST. En caso contrario, la limitación es más restrictiva, de 150 peticiones/hora. Este límite se renueva cada hora que pasa, volviéndose a reiniciar el contador de peticiones disponibles.

2.4.1. Recursos disponibles a través de la API REST

Timelines, TLs

Los *TLs* son colecciones de *tweets* ordenados del más reciente al más antiguo. Se pueden obtener los *tweets* publicados más recientemente por los *followings* de un usuario (`GET statuses/home_timeline`), los *tweets* más recientes en los que se menciona al usuario autenticado (`GET statuses/mentions`), los RTs más recientes publicados por el usuario autenticado (`GET statuses/retweeted_by_me`), los RTs más recientes que le han hecho al usuario autenticado (`GET statuses/retweets_of_me`), los *tweets* más recientes publicados por un usuario indicado mediante el parámetro `user_id` o el parámetro `screen_name`, que son su identificador o su nombre de usuario de *Twitter*, respectivamente (`GET statuses/user_timeline`), etc.

Tweets

Los recursos referentes a los *tweets* permiten obtener información sobre los mismos. En concreto existen recursos para obtener los 100 primeros usuarios que hacen RT de un determinado *tweet* (`GET statuses/:id/retweeted_by`), obtener los 100 primeros RTs de un *tweet* dado (`GET statuses/ /:id`), devolver la información de un *tweet* dado (`GET statuses/show/:id`), eliminar un *tweet* específico indicando su ID (`POST statuses/destroy/:id`), hacer RT de un *tweet* (`POST statuses/ retweet/:id`), publicar un *tweet* en el TL del usuario autenticado (`POST statuses/update`) indicando el texto del *tweet* en un parámetro requerido denominado `status`, etc.

Mensajes directos, DMs

Los DMs se pueden obtener a través de la REST API. Se puede acceder a los DMs más recientes del usuario autenticado (`GET direct_messages`), se pueden obtener los DMs más recientes que han sido enviados por el usuario que se encuentra autenticado (`GET direct_messages/sent`), se puede eliminar un DM del usuario autenticado indicando su ID (`POST direct_messages/destroy/:id`), enviar un nuevo DM a otro usuario (`POST direct_messages/new`) indicando al usuario destinatario mediante el parámetro `user_id` o el parámetro `screen_name`, y se puede mostrar la información acerca de un DM determinado indicando su ID (`GET direct_messages/show/:id`).

Followings y followers

Con la API se puede obtener información a cerca de los *followings* y los *followers* de un determinado usuario. Algunos de los recursos disponibles permiten obtener un array de IDs de los *followers* de un usuario dado (`GET followers/ids`), obtener un array de identificadores con los *followings* de un usuario dado (`GET friends/ids`), obtener relaciones entre *followings* de dos usuarios distintos dados (`GET friendships/exists`), permitir que el usuario autenticado siga a un usuario dado (`POST friendships/create`), permitir que el usuario autenticado deje de seguir a usuarios (`POST friendships/ destroy`), etc.

En todos los recursos anteriores en los que se necesita especificar a otro usuario que no sea el autenticado, es obligatorio indicar como parámetro o bien el `user_id` o bien el `screen_name` de dicho usuario.

Usuarios

Devuelve información sobre un usuario de *Twitter* dado mediante los parámetros `user_id` o `screen_name` (`GET users/show`), obtiene usuarios similares a uno dado mediante los parámetros `user_id` o `screen_name` (`GET users/search`), etc.

Usuarios sugeridos

Estos recursos acceden a recomendaciones de usuarios que otros usuarios pueden tener interés en seguir (`GET users/suggestions`, `GET users/suggestions/:slug`, `GET users/suggestions/:slug/members.format`). El `slug` indica el nombre de una lista o una categoría.

Favoritos

Los recursos sobre favoritos, obtienen información sobre *tweets* que han sido marcados como favoritos por un usuario determinado. Devuelven los *tweets* marcados como favoritos por un usuario indicado mediante los parámetros `user_id` o `screen_name` (`GET favorites`), marcan como favorito un *tweet* concreto indicado mediante su ID (`POST favorites/create/:id`) y desmarcan un *tweet* que haya sido marcado previamente como favorito dado su ID (`POST favorites/destroy/:id`).

Listas

Con estos métodos es posible obtener las listas a las que está suscrito el usuario autenticado (`GET lists/all`), obtener los últimos *tweets* de una lista especificada mediante el parámetro `list_id` (`GET lists/statuses`), borrar un miembro de una lista determinada especificada mediante el parámetro `list_id` (`POST lists/members/destroy`), devolver los usuarios suscritos a una lista especificada mediante el parámetro `list_id` (`GET lists/subscribers`), crear una nueva lista indicando su nombre mediante el parámetro `name` (`POST lists/create`), etc.

Cuentas

Estos recursos permiten obtener información sobre el límite de peticiones de una cuenta de *Twitter* (`GET account/rate_limit_status`), verificar las credenciales para la autenticación de un usuario (`GET account/verify_credentials`), finalizar la sesión del usuario que está autenticado (`POST account/end_session`), actualizar el perfil del usuario que está autenticado (`POST account/update_profile`), etc.

Notificationes

Estos recursos de la API REST permiten controlar las notificaciones que un usuario dado desea recibir mediante SMS, para ello existen métodos que habilitan o deshabilitan estas notificaciones (`POST notifications/follow`, `POST notifications/leave`).

En estos recursos el usuario se indica mediante el parámetro `user_id` o el parámetro `screen_name`.

Búsquedas salvadas

Estos recursos permiten al usuario autenticado guardar las referencias a los criterios de búsqueda para su posterior reutilización (`GET saved_searches`, `GET saved_searches/show/:id`, `POST saved_searches/create`, `POST saved_searches/destroy/:id`).

Lugares y localización

Estos métodos posibilitan conectar los datos de localización a los *tweets* para descubrir *tweets* y lugares (`GET geo/id/:place_id`, `GET geo/reverse_geocode`, `GET geo/search`, `GET geo/similar_places`, `POST geo/place`).

Tendencias

Estos recursos de la API permiten explorar los temas que son tendencia en *Twitter* (*trending topics*, TTs) (`GET trends/:woeid`, `GET trends/available`, `GET trends/daily`, `GET trends/weekly`).

Bloqueos

Estos métodos permiten bloquear o desbloquear usuarios en *Twitter*. El usuario autenticado puede bloquear a otro usuario indicado mediante los parámetros `user_id` o `screen_name` (`POST blocks/create`), puede desbloquear a otro usuario indicado mediante los parámetros `user_id` o `screen_name` (`POST blocks/destroy`), puede obtener una lista que contiene los usuarios que tiene bloqueados (con `GET blocks/blocking` o con `GET blocks/blocking/ids`), etc.

Informes de spam

Estos recursos permiten notificar cuentas de usuarios que son consideradas *spam*. El usuario considerado *spam* se indica mediante los parámetros `user_id` o `screen_name` al recurso `POST report_spam`.

OAuth

Twitter utiliza el protocolo *OAuth* para la autenticación de cuentas de usuario. Existen recursos que permiten a una aplicación solicitar un *token* temporal para el protocolo *OAuth* (`POST oauth/request_token`), obtener su *token* de acceso (`POST oauth/accesst_token`), y recursos que permiten a una aplicación consumidor usar *token* temporal para solicitar la autorización o la autenticación de un usuario (`GET oauth/authenticate`, `GET oauth/authorization`).

En el apartado 2.5 se explicará en detalle este protocolo.

Ayuda

Tal como indica su nombre, estos recursos REST ayudarán a los desarrolladores a depurar y a obtener información útil y de ayuda sobre la API de *Twitter*. Existen métodos que sirven para determinar la fecha de los servidores de *Twitter* (`GET help/test`), obtener datos de configuración de *Twitter* (`GET help/configuration`) y obtener una lista con los distintos idiomas soportados por *Twitter* junto con su código ISO 639-1 (`GET help/languages`).

Legal

Estos métodos de la API permiten obtener información sobre términos legales de *Twitter* (`GET legal/privacy`, `GET legal/tos`).

2.5. Protocolo OAuth

OAuth[13][14][15] es un protocolo abierto que permite una autorización segura de una API para aplicaciones de escritorio, móviles y web.

Con este protocolo, los desarrolladores de servicios que quieren acceder a cuentas externas, pueden interactuar y publicar datos protegidos a la par que proporcionar a los usuarios un acceso seguro a sus datos, protegiendo las credenciales de sus cuentas.

OAuth permite a un usuario de un sitio A (proveedor de servicio) compartir su información en el sitio A con otro sitio B (consumidor), pero sin necesidad de compartir toda su identidad.

En el proceso *OAuth* intervienen tres actores, que son:

1. Proveedor de servicios, o lo que es lo mismo, el servicio de autenticación externo (en este caso *Twitter*).
2. Consumidor, que es al servicio al que el usuario quiere acceder usando una cuenta externa (la del proveedor de servicios).
3. Usuario.

Twitter en su API posee métodos para utilizar este protocolo en las aplicaciones desarrolladas por terceros. Es importante distinguir entre la autorización y la autenticación, ambas acciones se pueden realizar a través de la REST API. La autorización se centra en un escenario en el que el consumidor quiere acceder a información del usuario en el proveedor de servicios y el usuario autoriza dicho acceso. La autenticación, en cambio, permite al usuario acceder a información y recursos en el consumidor, pero autenticarse con su cuenta del proveedor de servicios. En el primer caso, se otorga permiso para realizar una acción, mientras que en el segundo, se valida la identidad de un usuario.

Con *OAuth*, los pasos que hay que realizar para la autenticación y la autorización son los mismos. En el caso de la API de *Twitter* lo único que cambiaría son las llamadas a las URLs. Para la autorización se utiliza el método `GET oauth/authorize`, mientras que para la autenticación se usa `GET oauth/authenticate`.

Flujo de trabajo de OAuth en Twitter

Previamente a la autenticación con *OAuth*, debe haberse dado de alta a la aplicación consumidor en *Twitter*. Si es así, la aplicación dispondrá de un `consumer_key` y un `consumer_secret`, parámetros necesarios en el flujo de trabajo con *OAuth*.

Cómo se registra una aplicación en *Twitter* se explica en el apéndice A, apartado A.1.2.

La autenticación con *OAuth* se realiza en tres pasos:

1. El consumidor obtiene un *token* temporal.
2. Después, el usuario autoriza el *token*.
3. Por último, se intercambia el *token* temporal por un *token* de acceso permanente.

Paso 1. Obtención del token temporal

El consumidor solicita un *token* temporal al proveedor de servicio de manera transparente para el usuario. El único propósito de este *token* es recibir la aprobación del usuario, solamente se usa para poder obtener posteriormente el *token* de acceso permanente. Para solicitar un *token* temporal a *Twitter*, el consumidor necesitará enviar los siguientes parámetros en la petición:

- `oauth_consumer_key`: cuando un desarrollador da de alta una aplicación en *Twitter*, se obtiene este dato.
- `oauth_signature_method`: indica el método empleado para generar la firma. En el caso de *Twitter* hay que usar HMAC-SHA1.

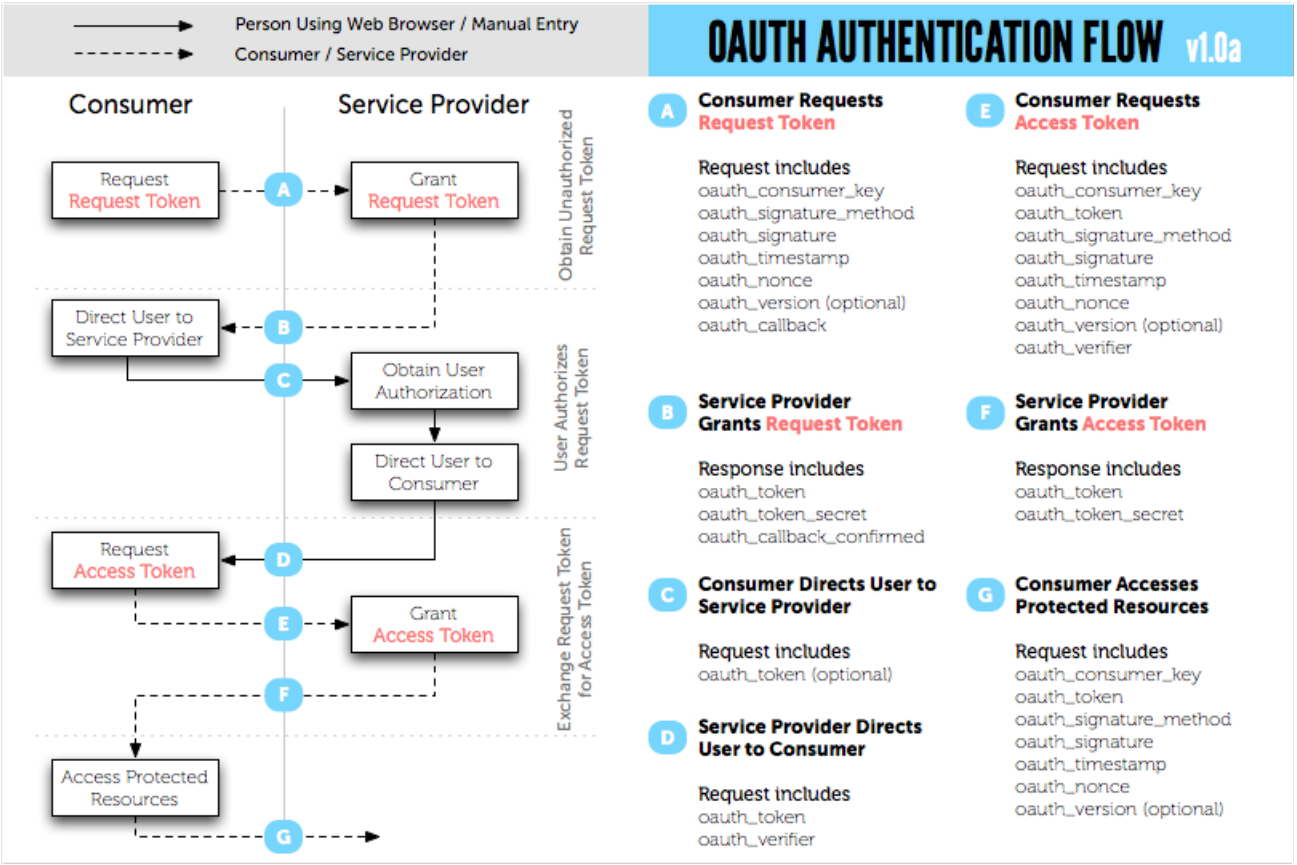


Figura 2.8: Flujo de trabajo para OAuth

- `oauth_signature`: es la firma generada por la aplicación dada de alta (consumidor).
- `oauth_nonce`: cadena aleatoria que acompaña a un *timestamp*.
- `oauth_timestamp`: *timestamp*, comprendido entre el 1 de enero de 1970 y la fecha actual. Se usa por el servidor junto al `oauth_nonce` para verificar que la petición nunca se ha realizado antes y prevenir ataques.
- `oauth_version` (opcional): en la actualidad se está utilizando la versión 1.0 de *OAuth*.
- `oauth_callback`: como se ha mencionado anteriormente, *OAuth* puede usarse en aplicaciones de escritorio y en aplicaciones web. Este parámetro es importante para que el servicio se redirija a esta dirección de *callback* tras la autorización de acceso. En *Twitter* este parámetro se puede omitir si la aplicación es de escritorio, en caso contrario hay que indicarlo.

  La petición será un POST a la URL `https://api.twitter.com/oauth/request_token` con los parámetros indicados en una cabecera de autorización.

Como respuesta, si todo va bien, el proveedor de servicio otorga el *token* temporal al consumidor. En esta respuesta se pueden encontrar el `oauth_token` (*token* temporal) y el `oauth_token_secret` (*token* secreto).

Paso 2. Obtención de la autorización del usuario

El consumidor no puede usar el *token* temporal obtenido hasta que el usuario lo autorice. Para ello:

1. El consumidor redirige al usuario a una página segura en el proveedor de servicio. Para que el consumidor intercambie el *token* temporal por un *token* de acceso, éste debe obtener aprobación del usuario redirigiéndole a una página segura en el proveedor de servicios. El consumidor debe hacer una petición HTTP a `https://api.twitter.com/oauth/authorize` (si se trata de autorización), o a `https://api.twitter.com/oauth/authenticate` (si se trata de autenticación), sin olvidar indicar el `oauth_token` y el `oauth_callback` como parámetros de la URL (opcional).
2. Una vez en la página segura del proveedor de servicio, el usuario se debe autenticar y dar los permisos necesarios para que el consumidor pueda acceder a su información.
3. Tras esto, el consumidor debe ser notificado de que el *token* temporal ha sido autorizado ya para cambiarse por un *token* de acceso. Si el usuario hubiese denegado el acceso, esto también puede notificarse al consumidor. Si el consumidor ha indicado previamente el parámetro `oauth_callback`, el proveedor de servicio redirigirá al usuario a la URL indicada, pasando como parámetro el `oauth_token` (*token* temporal) obtenido anteriormente.

   Si no hubiese sido especificada ninguna URL de *callback*, como ocurre en el caso de aplicaciones de escritorio, el proveedor de servicio comunica al usuario que informe manualmente al consumidor de que la autorización se ha completado.

Paso 3. Obtener el token de acceso

El consumidor intercambia el *token* temporal por un *token* de acceso capaz de acceder a los recursos protegidos del usuario en el proveedor de servicio:

1. Para solicitar un *token* de acceso permanente, el consumidor debe realizar una petición HTTP al proveedor de servicio a la URL `http://twitter.com/oauth/access_token`, pasando como parámetros:

   - `oauth_consumer_key` del consumidor.
   - `oauth_token`: en este caso, se refiere al token temporal obtenido.
   - `oauth_signature_method`.
   - `oauth_signature`.
   - `oauth_timestamp`.
   - `oauth_nonce`.
   - `oauth_version` (opcional): si se indica debe ser 1.0.

2. El proveedor de servicio garantiza un *token* de acceso que estará formado por un nuevo `oauth_token` y un `oauth_token_secret`. Estos datos asociados al usuario, deben ser almacenados por el consumidor y usados cuando se requiera acceder a recursos protegidos del proveedor de servicios.

Accediendo a recursos protegidos

Una vez conseguido el *token* de acceso, el consumidor podrá acceder a los recursos protegidos del proveedor de servicios en nombre del usuario. Para ello, todas las peticiones deben estar firmadas y contener una cabecera de autorización con los parámetros:

- `oauth_consumer_key`.
- `oauth_token`: token de acceso.
- `oauth_signature_method`.
- `oauth_timestamp`.
- `oauth_nonce`.
- `oauth_version` (opcional).
- cualquier parámetro adicional que se definan en el proveedor de servicio.

2.6. CSS

El CSS[16][17] (*Cascading Style Sheets*) es un lenguaje usado para definir la parte de presentación de un documento estructurado. Dicho documento puede estar escrito en HTML o XML. El W3C (*World Wide Web Consortium*) se encarga de elaborar las especificaciones y los estándares de las hojas de estilo.

El CSS surgió con el propósito de separar la estructura de un documento de su presentación. Gracias a ello, se ha permitido eliminar el uso de tablas para la maquetación, usándolas únicamente para la muestra de datos tabulados.

2.6.1. Sintaxis

CSS tiene una sintaxis sencilla que usa palabras clave que especifican los nombres de los selectores, propiedades y atributos de los que quiere modificar la presentación.

En estas hojas se definen una serie de reglas que comprenden uno o más selectores y un bloque de código con los estilos que se desean aplicar en los elementos del documento estructurado que cumplan con el selector que se indica. Cada uno de estos bloques se encierra entre llaves, y dentro de ellas se declaran una o varias propiedades de estilo con el formato `propiedad:valor`.

Los selectores a los que se hace referencia en cada bloque del CSS, indicarán los elementos que se verán afectados en función de su tipo, nombre (*name*), ID, clase (*class*), posición dentro del DOM (*Document Object Model*), etc.

2.6.2. Uso

Para dar formato a un documento se puede utilizar el CSS de tres formas distintas:

1. A través de una hoja de estilo interna incrustada dentro del documento. Se debe definir dentro del elemento `<head>` de un documento HTML, marcada por la etiqueta `<style>`. De este modo se separa la información de estilo del código HTML.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN">
<html>
  <head>
    <title>hoja de estilo interna</title>
    <style type="text/css">

      body {
        padding-left: 11em;
        font-family: Georgia, "Times New Roman", serif;
        color: red;
        background-color: #d8da3d;
      }

      h1 {
        font-family: Helvetica, Geneva, Arial, sans-serif;
      }

    </style>
  </head>
  <body>
    <h1>Aqui se aplicara el estilo de letra para
    el titulo</h1>
  </body>
</html>
```

2. A través de una hoja de estilo externa. En este caso, el código CSS se escribirá en un fichero con extensión `.css` externo al documento. Este modo es el más potente, puesto que separa totalmente las reglas de estilo de la estructura del documento. La hoja de estilo se asociará al documento a través del elemento `<link>`, que debe situarse en la sección `<head>`.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN">
<html>
  <head>
    <title>Titulo</title>
    <link rel="stylesheet" type="text/css"
    href="http://www.w3.org/css/officeFloats.css" />
  </head>
  <body>
  .
  .
  .
  .
  </body>
</html>
```

3. Utilizando estilos directamente sobre los elementos del documento que lo permiten a través del atributo `<style>` dentro de la etiqueta `<body>`. Este tipo de definición de estilo pierde las ventajas que ofrece el CSS, ya que se mezcla el contenido con la presentación.

2.6.3. Ejemplos y normas básicas

En el ejemplo siguiente se indica que el selector `h1` de un documento HTML se verá afectado por la regla de estilo que va en su interior.

```
h1{color: red;}
```

Los selectores pueden aparecer de manera individual o separados por comas, lo que posibilita que las reglas de estilo afecten a varios elementos a la vez. Ejemplo:

```
h1, h2, h3 {
  color: red;
}
```

La propiedad color del ejemplo especifica el aspecto a cambiar. Las propiedades que se desean modificar en un CSS para un mismo selector pueden agruparse, pero será necesario separar cada una de ellas con un punto y coma, tal y como se muestra a continuación.

```
h1 {
  padding-left: 11em;
  font-family: Georgia, "Times New Roman",Times, serif;
  color: red;
  background-color: #d8da3d;
}
```

Si el valor que toma la propiedad está formado por más de una palabra, hay que ponerlo entre comillas.

```
p {font-family: "sans serif";}
```

Capítulo 3

Requisitos de diseño

En este capítulo se enumerarán los requisitos previos que se definieron antes del desarrollo de la aplicación:

1. La aplicación será web y, por tanto, correrá en un servidor.
2. El usuario que desee usar la aplicación tendrá que disponer de dos cuentas de *Twitter*: una cuenta robot y una cuenta administradora. También es válido que la cuenta robot coincida con la cuenta administradora, si se desea esto, con disponer de una cuenta de *Twitter* sería suficiente.
3. Un administrador puede administrar solamente una cuenta robot.
4. Una cuenta robot puede tener un único administrador.
5. Un usuario que quiera hacer uso del sistema, tendrá que darse de alta a través de la interfaz web de la aplicación. Esto lo hará completando un formulario de registro en el que deberá indicar su nombre de usuario (que debe ser el mismo que el de la cuenta de *Twitter* que elija como cuenta administradora), una contraseña para autenticarse en el sistema, y un correo electrónico.
6. El sistema podrá albergar a múltiples usuarios que deseen utilizar el servicio. Es decir, habrá tantas cuentas robot como usuarios administradores se registren en el sistema. Es importante remarcar que la asociación entre administrador y cuenta robot es 1:1.
7. Cuando un administrador se autentique por primera vez en el sistema tras haberse dado de alta, tendrá que asociar una cuenta robot a su cuenta. Para ello, éste será redirigido a una página oficial y segura de *Twitter*. Esta página pedirá permisos al usuario para que la aplicación pueda acceder a los recursos protegidos de la cuenta robot en *Twitter* sin necesidad de comprometer la seguridad de la cuenta. Esto se hará mediante el protocolo *OAuth*. A través de este protocolo la aplicación tendrá permisos para acceder a la información de la cuenta robot en *Twitter* (podrá hacer RTs en su *timeline*, leer DMs, etc.).
8. Una cuenta robot, para funcionar como tal, debe configurarse asociando a ella una serie de usuarios de *Twitter* (cuentas a agregar a la comunidad). Éstos serán los usuarios de los que la cuenta pueda hacer RT de manera automática. Los usuarios serán introducidos por el administrador a través de un formulario de configuración disponible en su interfaz web.
9. El máximo de usuarios que se permite que un administrador agregue a la comunidad de una cuenta robot es de 1000.
10. La cuenta robot no puede estar entre los usuarios que el administrador agregue a la comunidad. Esto es así porque una cuenta de *Twitter* no puede hacerse RT a sí misma.

11. Se definirán una serie de comandos propios de la aplicación para que el administrador los ejecute. Podrán ejecutarse o bien desde una consola de comandos disponible en la interfaz web del administrador, o bien enviado DMs a la cuenta robot desde la cuenta administradora. Si se elije la segunda opción, el contenido de los DMs será el comando en cuestión. Estos comandos le servirán al administrador para configurar la cuenta robot y gestionar lo que quiere que se publique de manera automática en la misma.

    Existen cuatro tipos de comandos. Tres que pueden ser ejecutados por un administrador y uno que puede ser ejecutado por los usuarios que forman parte de la comunidad de una cuenta robot:

    - `ADD`. Este comando puede ejecutarse únicamente por un administrador de una cuenta robot (desde la consola de la interfaz web o por medio de DMs). Sirve para añadir instrucciones de seguimiento que una cuenta robot comprobará periódicamente. Estas instrucciones indican el contenido que resulta relevante para el administrador, o lo que es lo mismo, el contenido publicado por los usuarios de su comunidad que quiere que se publique de modo automático en la cuenta robot (*hashtags* relevantes para casa usuario de la comunidad). Formato:
        - `add -u user1, user2..., userN -q #hashtag1, #hashtag2..., #hashtagM`
        - `add -u * -q #hashtag1, #hashtag2..., #hashtagM`
        - `add -u user1, usert2..., userN -q *`
    - `DELETE`. Al igual que la instrucción anterior, puede ser ejecutada solo por un usuario que sea administrador (desde la consola de la interfaz web o por medio de DMs). Sirve para eliminar instrucciones previamente añadidas a una cuenta robot. Formato:
        - `delete -u user1, user2..., userN -q #hashtag1, #hashtag2..., #hashtagM`
        - `delete -u * -q #hashtag1, #hashtag2..., #hashtagM`
        - `delete -u user1, usert2..., userN -q *`
    - `UPDATE`. También es una instrucción exclusiva de un administrador de una cuenta robot, pero en este caso, esta instrucción solo puede ejecutarse vía consola de comandos de la interfaz web. Sirve para comprobar, de manera síncrona, si usuarios concretos de la comunidad de la cuenta robot han actualizado su estado en *Twitter*. En ese caso, se comprobará si el *tweet* o los *tweets* que estos usuarios hayan publicado resultan de interés para el administrador de la cuenta robot (es decir, contienen alguno de los *hashtags* relevantes definidos en las instrucciones que estén activas). Si esto ocurre, estos *tweets* serán almacenados para que, posteriormente, la cuenta robot haga RT de todos ellos. Formato:
        - `update -u user1, user2..., userN`
        - `update -u *`
    - `SUGGESTION`. Los comandos con dicha instrucción pueden ser ejecutados únicamente por usuarios que formen parte de la comunidad de una cuenta robot. Sirven para enviar a la cuenta robot, a través de DMs, sugerencias de nuevas instrucciones de seguimiento.
        - `suggestion -u user1, user2..., userN -q #hashtag1, #hashtag2..., #hashtagM`
        - `suggestion -u * -q #hashtag1, #hashtag2..., #hashtagM`
        - `suggestion -u user1, usert2..., userN -q *`

12. Cada cierto tiempo la aplicación actualizará automáticamente de manera asíncrona todas las cuentas robots que estén dadas de alta en el sistema. Este intervalo de tiempo entre actualizaciones se

calculará, para cada cuenta, en función al número de usuarios que tenga agregados. En cada cuenta robot, se comprobará si cada uno de sus usuarios agregados ha publicado nuevos *tweets*. En caso de ser así, se comprobará si dichos *tweets* resultan de interés y, si es así, todos ellos serán almacenados para *retwittear* posteriormente.

13. Estas comprobaciones también las podrá forzar el administrador de manera síncrona a través de un botón denominado *«Forzar actualización»* disponible en su interfaz web. Este botón comprobará los *timelines* de los usuarios agregados a la cuenta robot en busca de *tweets* relevantes que serán almacenados.

14. Mediante otro botón denominado *«Forzar RTs pendientes»*, el administrador podrá forzar los RTs pendientes de los *tweets* que se hayan almacenado en el paso previo.

15. Cada cierto tiempo, para cada cuenta robot se comprobará si se han recibido nuevos DMs válidos que contengan comandos a ejecutar. Si es así, estos comandos se ejecutarán.

16. La comprobación de nuevos DMs recibidos también podrá realizarse a través de un botón denominado *«Comprobar DMs»* disponible en la interfaz web del administrador.

17. Desde la interfaz web de la aplicación, en una pestaña denominada *«Sugerencias»*, se le indicarán al administrador las cinco sugerencias más prioritarias enviadas por los usuarios y las cinco sugerencias de seguimiento más prioritarias calculadas de manera automática por el sistema. Se debe tener en cuenta que:

    - El sistema sugiere en base a lo que publiquen los usuarios seguidos por una cuenta robot. Tiene en cuenta tanto las menciones que aparezcan en los *tweets*, como los *hashtags*. Por ello puede recomendar seguir un *hashtag* determinado tanto para usuarios que ya estén agregados a la comunidad de la cuenta robot, como para usuarios que no lo estén aún.
    - Los usuarios pueden mandar todo tipo de sugerencias (siempre que respeten el formato del comando `SUGGESTION`).

    Cada una de estas sugerencias podrá ser añadida por el administrador a través de un enlace denominado *«Añadir»* que se encontrará junto a cada una de ellas. También podrá ser descartada pulsando sobre un enlace denominado *«Ignorar»*. Las sugerencias se mostrarán por orden de prioridad, y la prioridad se medirá por la frecuencia de aparición de las mismas.

    Si el administrador intenta agregar una sugerencia para un usuario al que aún no siga la cuenta robot, el usuario no perteneciente a la comunidad será añadido a ésta automáticamente por el sistema. Además, después se agregará su *hashtag* correspondiente a la lista de instrucciones activas para la cuenta robot.

18. Todas las sugerencias (tanto las del sistema como las de los usuarios), se le enviarán diariamente al administrador a su correo electrónico de modo informativo. En el correo se indicarán las cinco sugerencias más relevantes enviadas por los usuarios y las cinco sugerencias más relevantes calculadas por el sistema. Si no hay ninguna, no se enviará correo.

19. Las sugerencias acumuladas que no hayan sido agregadas por el administrador, tendrán caducidad de una semana (a no ser que vuelvan a aparecer de nuevo y se renueve su fecha de actualización).

20. Todos los errores que se produzcan en la interacción del administrador con la interfaz web de la aplicación, le serán informados a éste mediante mensajes de advertencia (posibles errores son: datos erróneos al rellenar formularios, instrucciones a ejecutar con sintaxis incorrecta, etc.).

Capítulo 4

Diseño de alto nivel

En este capítulo se hablará del diseño de alto nivel de la aplicación desarrollada. Se detallarán las pantallas que forman parte de la interfaz web del administrador de la cuenta robot, indicando las distintas situaciones que se pueden producir en la interacción del usuario con la interfaz web.

4.1. Acceso a la aplicación y pantalla de inicio

Suponiendo que la aplicación desarrollada está corriendo en un servidor web, se podrá acceder a ella introduciendo la URL `http://localhost:8080/tuiter-bot/` en un navegador. Nótese que se está suponiendo que la aplicación se ha instalado en un servidor local, en el caso de que se encuentre en un servidor externo, en el navegador habría que introducir una URL con el formato `http://{servidor}:{puerto}/tuiter-bot/`.

Si todo va bien se abrirá la página inicial de la aplicación, cuyo esquema de diseño será una pantalla similar a la que se muestra en la figura 4.1.

4.1.1. Pestaña «Manual de usuario»

Al hacer clic sobre esta pestaña se abrirá una página que contendrá las instrucciones de uso de la aplicación.

4.1.2. Pestaña «Contacto»

Si el usuario pulsa sobre esta pestaña, se abrirá una página que contendrá una dirección de correo electrónico de contacto para que futuros usuarios expongan sus dudas o sugerencias sobre la aplicación.

4.1.3. Pestaña «Alta administrador»

Al hacer clic en esta pestaña, el usuario podrá darse de alta como administrador de una cuenta robot para hacer uso de la aplicación y de sus funcionalidades (ver figura 4.2).

Tendrá que introducir los datos que se requieren en el formulario disponible (cuenta de *Twitter* elegida como cuenta administradora, contraseña para autenticarse en el sistema, y dirección de correo electrónico del administrador).

Podrán producirse dos situaciones cuando el administrador se registre en el sistema:

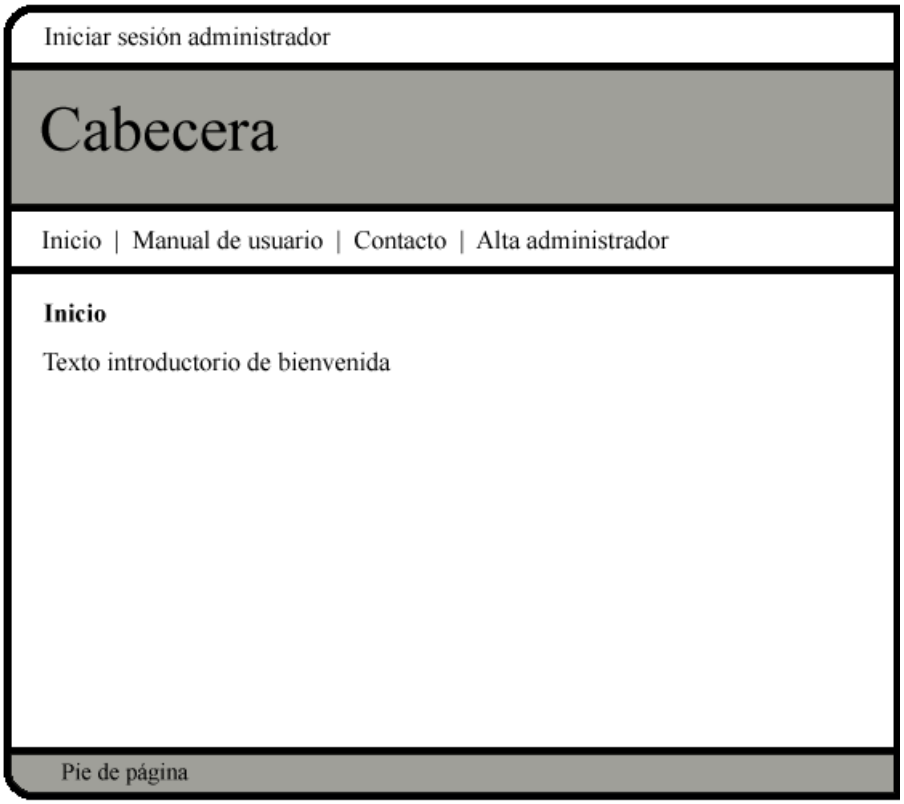


Figura 4.1: Diseño página de inicio

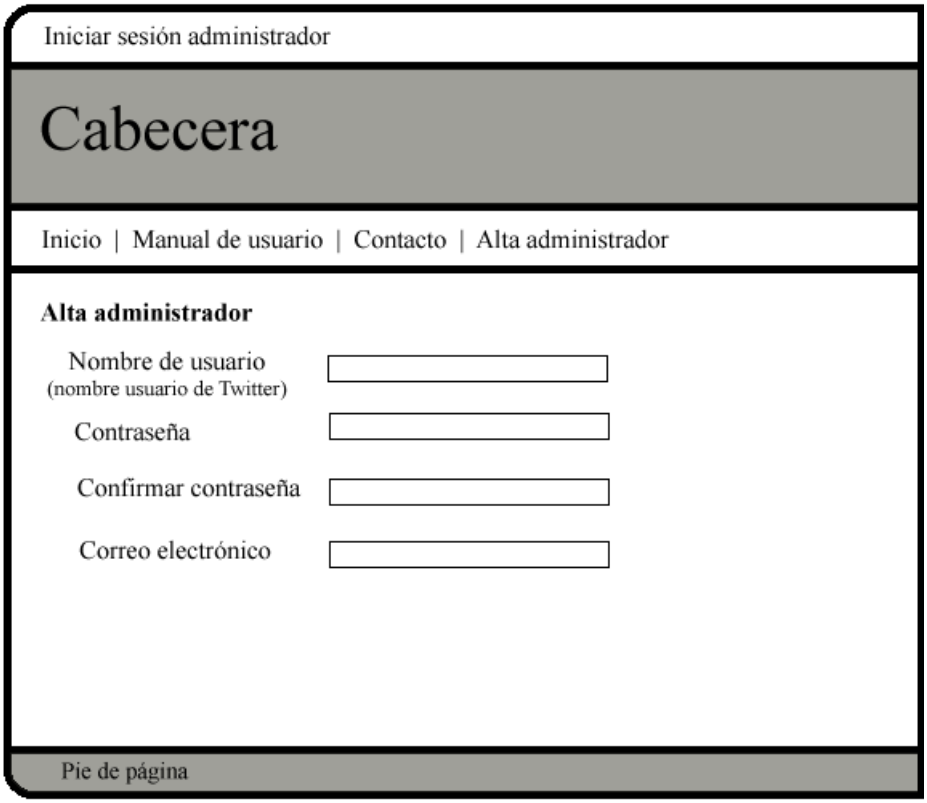


Figura 4.2: Diseño página de formulario de alta administrador

1. Que se introduzca algún dato erróneo en el formulario y el registro no se lleve a cabo. Si esto ocurre, la situación le será notificada al usuario mediante un mensaje de advertencia. Los errores pueden producirse por dejar campos vacíos, por indicar un nombre de usuario que esté ya registrado, por introducir un nombre de usuario o una cuenta de correo con formato incorrecto, etc.

2. Que el usuario se registre de manera correcta en el sistema, en cuyo caso será informado mediante un mensaje de que el proceso de registro se ha llevado a cabo de manera satisfactoria.

4.2. Autenticación y pantallas del administrador

4.2.1. Autenticación del administrador

Para entrar en el sistema, el usuario deberá hacer clic sobre la opción *«Iniciar sesión administrador»* ubicada en la parte superior de la pantalla (ver figura 4.1).

A continuación se le mostrará una nueva pantalla que contendrá un formulario en el que introducir las credenciales de la cuenta del administrador que haya sido dada de alta previamente (ver figura 4.3). Se podrán producir dos situaciones:

1. Si las credenciales introducidas son incorrectas, el error le será notificado al usuario mediante un mensaje de advertencia.

2. En caso contrario, se iniciará la sesión del administrador en el sistema.

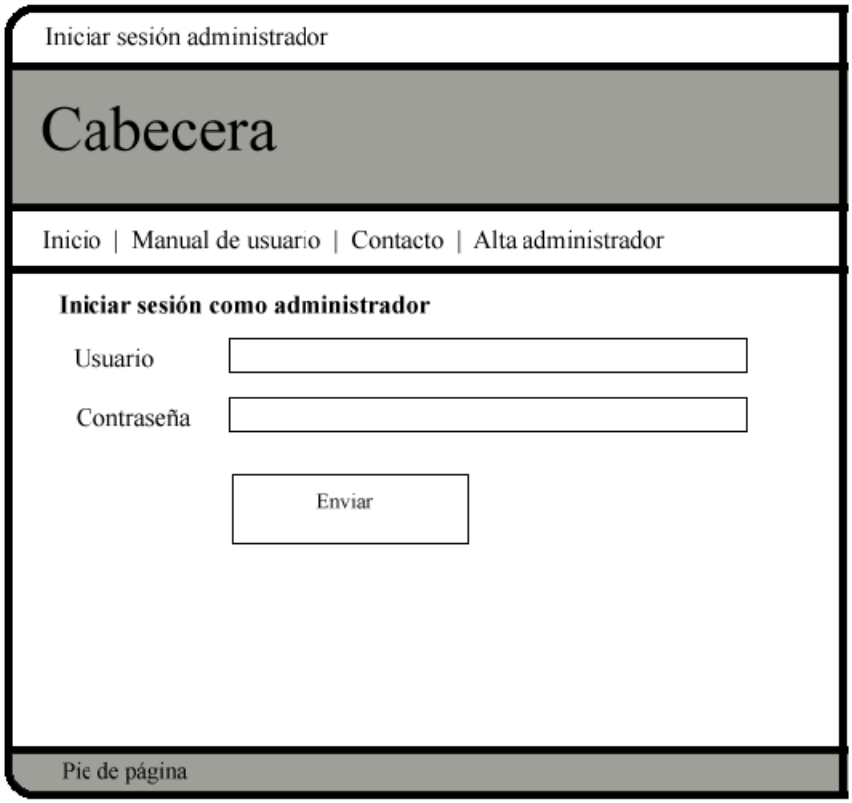


Figura 4.3: Diseño de la página de inicio de sesión

Tras el inicio de sesión del administrador en el sistema podrán darse dos casos:

1. Es la primera vez que se autentica en la aplicación o no hay asignada aún a su cuenta de administrador ninguna cuenta robot. En este caso, el administrador será redirigido a una página oficial de *Twitter* en la que deberá introducir las credenciales (usuario y contraseña) de la cuenta de *Twitter* que quiera que haga las labores de cuenta robot.

    - Si el administrador introduce correctamente las credenciales de la cuenta robot y acepta los permisos que se requieren desde la página de *Twitter*, éste será redirigido de nuevo a la página de inicio de la interfaz web de la aplicación (con su sesión abierta, ver figura 4.4). Automáticamente quedará asignada a la cuenta del administrador una cuenta robot.
    - Si el administrador introduce de manera errónea las credenciales de la cuenta robot o deniega los permisos que se solicitan desde la página de *Twitter*, éste será redirigido a la página de inicio de la interfaz web de la aplicación con su sesión abierta. Además se le notificará mediante un mensaje de advertencia que aún no tiene asignada ninguna cuenta robot.

2. El segundo caso se da cuando el administrador ya tiene asignada una cuenta robot a su cuenta. Tras el inicio de sesión no se le redirigirá a ninguna página de *Twitter*, sino que directamente será enviado a la página principal de la interfaz web de la aplicación con su sesión abierta (ver figura 4.4).

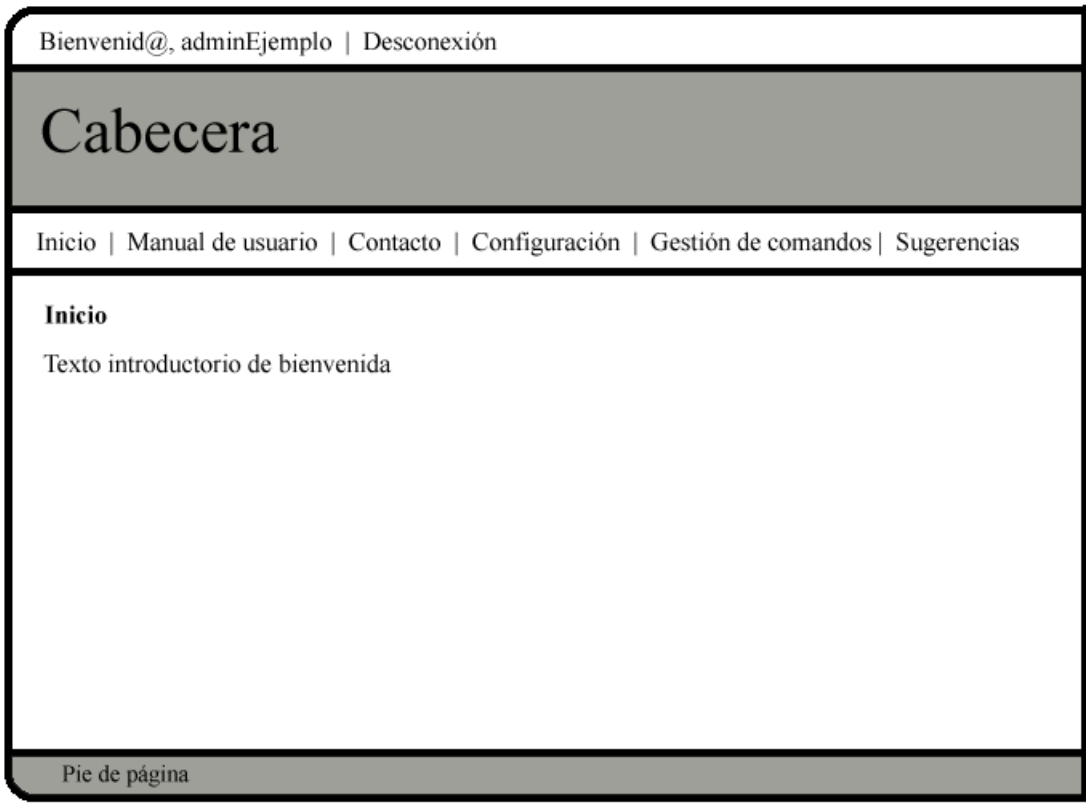


Figura 4.4: Diseño de la página de inicio del administrador

4.2.2. Pestaña «Configuración»

Al hacer clic en esta pestaña, solo disponible una vez que se ha iniciado sesión en el sistema, aparecerá una nueva pantalla con dos opciones a elegir: *«Ajustes de configuración»* y *«Detalles de configuración»* (ver figura 4.5).

Enlace de «Ajustes de configuración»

Si se pulsa sobre *«Ajustes de configuración»* aparecerá una nueva pantalla que contiene un formulario. Rellenando dicho formulario el administrador podrá configurar los nombres de las cuentas de *Twitter* que desea agregar a su cuenta robot (ver figura 4.6).

Se observa que en el formulario aparecerán tres campos. Los dos primeros, el nombre de usuario y el administrador, no son editables. Son valores fijos que indican los nombres de usuario de *Twitter* del administrador y de la cuenta robot. El tercer campo es el que habrá que configurar, e indicará los usuarios que se desea que la cuenta robot siga.

Si el usuario pulsa sobre el botón *«Enviar»* pueden darse dos situaciones:

1. Que se produzca un error. El error podría deberse a que se haya dejado el campo de usuarios en blanco, se haya indicado algún usuario con un formato incorrecto, o se haya agregado como usuario a la propia cuenta robot. En este caso, el error producido se le notificará al administrador mediante un mensaje de advertencia.

2. Que se guarden los cambios realizados en la configuración correctamente. En este caso el administrador será redirigido a la página de *«Detalles de configuración»* (ver figura 4.7).

Existirá también un botón, situado junto al de *«Enviar»*, denominado *«Confirmar permisos»*. Su función será la de posibilitar reconfigurar nuevamente una cuenta robot. Redirigirá al administrador a la página de *Twitter* que aparece la primera vez que éste inicia sesión en el sistema. Dicho botón servirá, por tanto, para que si un usuario quiere modificar su cuenta robot asociada lo pueda hacer a través de la interfaz.

Enlace de «Detalles de configuración»

Si se pulsa sobre este enlace, el administrador accederá a una pantalla en la que se mostrarán los datos de configuración de la cuenta robot: nombre de usuario del administrador, nombre de usuario de la cuenta robot, nombres de usuarios de *Twitter* a los que seguirá la cuenta robot e intervalo de comprobación (que especifica el tiempo entre intervalos de comprobación asíncrona, se explicará con más detalle en posteriores capítulos) (ver figura 4.7).

4.2.3. Pestaña de «Gestión de comandos»

Si el administrador accede a esta pestaña se le mostrará una pantalla en la que aparecerán dos enlaces asociados a la gestión de comandos (ver figura 4.8).

Enlace de «Ejecutar comandos»

Al pulsar sobre este enlace, el administrador accederá a una pantalla en la que dispondrá de una consola para ejecutar los comandos propios de la aplicación (`ADD`, `DELETE` y `UPDATE`) mediante el botón *«Ejecutar comando»* (ver figura 4.9).

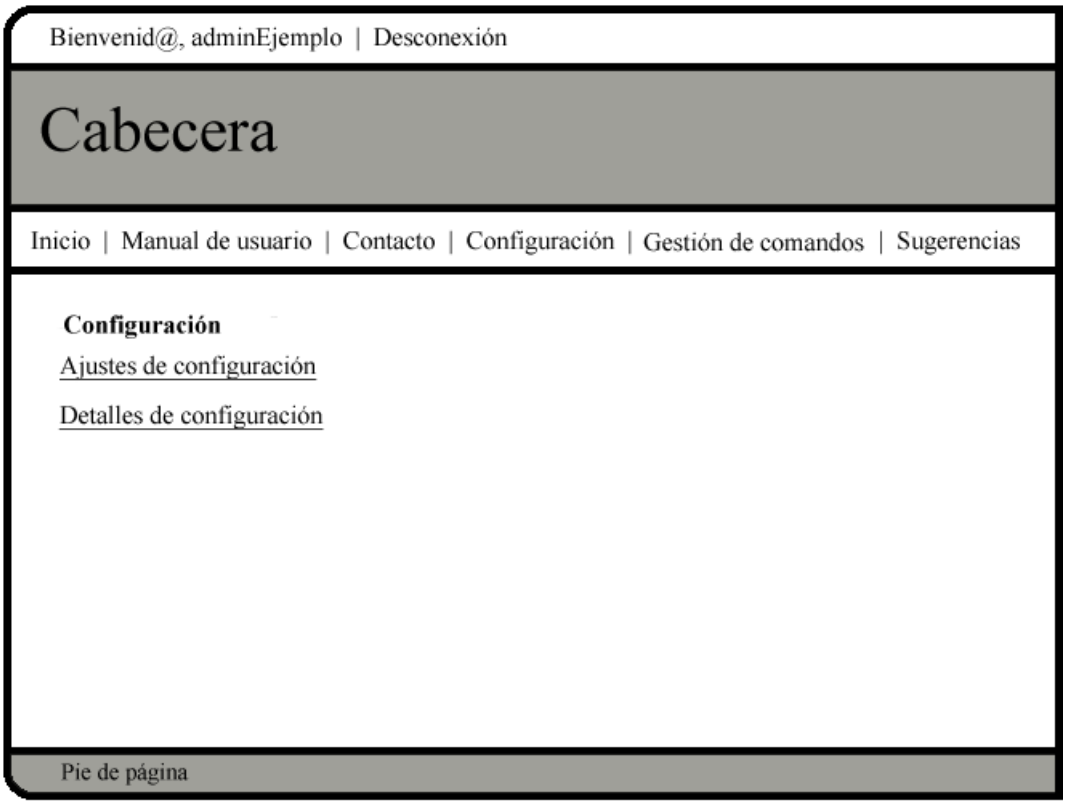


Figura 4.5: Diseño de la página del menú de configuración

La interfaz indicará en un texto informativo el número de peticiones disponibles a la API de *Twitter*. Esto es así porque, tal y como se ha mencionado en capítulos anteriores, las peticiones a la API de *Twitter* tienen un límite por hora. Además, la aplicación reparte este límite de peticiones de manera que solo se reserven 60 a la hora para ejecutar de forma síncrona (desde interfaz).

La pantalla también dispondrá de otros tres botones. El botón de *«Comprobar DMs»*, tal y como indica su nombre, servirá para comprobar si en la cuenta robot se han recibido nuevos DMs cuyos remitentes sean el administrador o los usuarios seguidos. El contenido de estos DMs, para que sean válidos, debe ser un comando de tipo `SUGGESTION` (en el caso de que el remitente sea alguno de los usuarios seguidos por la cuenta robot) o un comando de tipo `ADD` o `DELETE` (en el caso de que lo haya enviado el administrador). Si hay DMs nuevos y cumplen las reglas especificadas, estos comandos serán ejecutados internamente por la aplicación. La ejecución del comando `UPDATE` no está permitida mediante DM. Esto se debe a que esta instrucción está pensada para ejecutarse únicamente de manera síncrona, ya que su función es forzar la actualización de los usuarios indicados en el instante en el que se llama al comando, y carece de sentido tener que esperar a que se comprueben los DMs para que se ejecute.

El botón *«Forzar actualización»* servirá para forzar la actualización de una cuenta robot. Es decir, chequear uno por uno a los usuarios agregados a una cuenta robot y comprobar si cada uno de ellos ha publicado algún *tweet* que cumpla alguna de las instrucciones que estén activas para esa cuenta robot (contengan *hashtags* relevantes). Si hay *tweets* relevantes, la cuenta robot hará RT de todos ellos.

El último botón disponible, *«Forzar RTs pendientes»*, hará que la cuenta robot haga RT de todos aquellos *tweets* que se hayan marcado como relevantes en alguna actualización y no hayan podido ser *retwittea-*

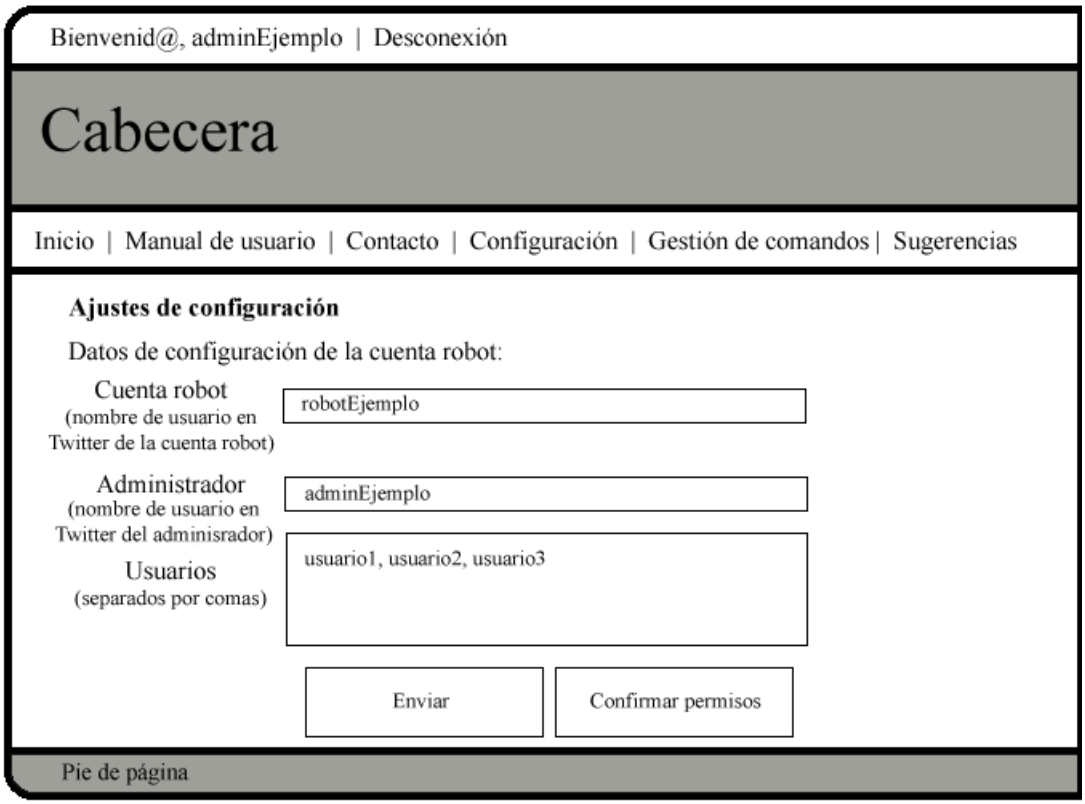


Figura 4.6: Diseño de la página de ajustes de configuración

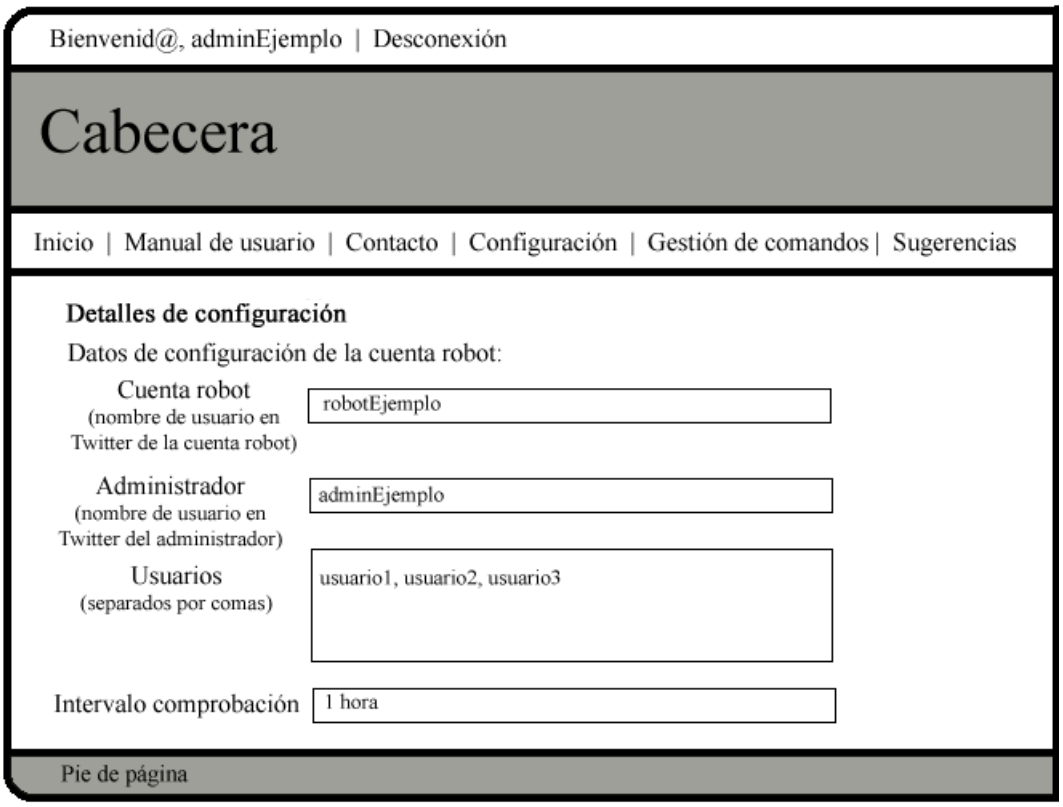


Figura 4.7: Diseño de la página de detalles de configuración

dos aún. Bien debido a escasez de peticiones disponibles a la API, o a posibles caídas momentáneas del servidor de *Twitter*. Si no hay RTs pendientes y se pulsa sobre el botón, se indicará la situación mediante un mensaje.

Los casos de error que se pueden producir en esta pantalla de la interfaz son:

- Que el administrador introduzca un comando con un formato incorrecto y lo ejecute.
- Que el administrador ejecute comando habiendo dejado en blanco el contenido de la consola.
- Que se ejecute algún comando, la actualización de usuarios, la actualización de DMs o el forzado de RTs, habiéndose acabado las peticiones disponibles a la API de *Twitter* en ese instante.

En todos estos casos se le notificará al usuario el error mediante un mensaje de advertencia.

Enlace de «Ver instrucciones activas»

En esta pantalla se visualizarán todas las instrucciones que hayan sido activadas por el administrador para su cuenta robot. Cada una de estas instrucciones estará formada por un usuario (perteneciente a los agregados a la cuenta robot) seguido de un conjunto de *hashtags*. Ello significará que cuando se compruebe si el usuario ha publicado nuevos contenidos en su cuenta de *Twitter*, si ha escrito *tweets* que contengan alguno de los *hashtags* indicados, la cuenta robot hará RT de los mismos.

La pantalla de instrucciones activas tiene el formato que se muestra en la figura 4.10.

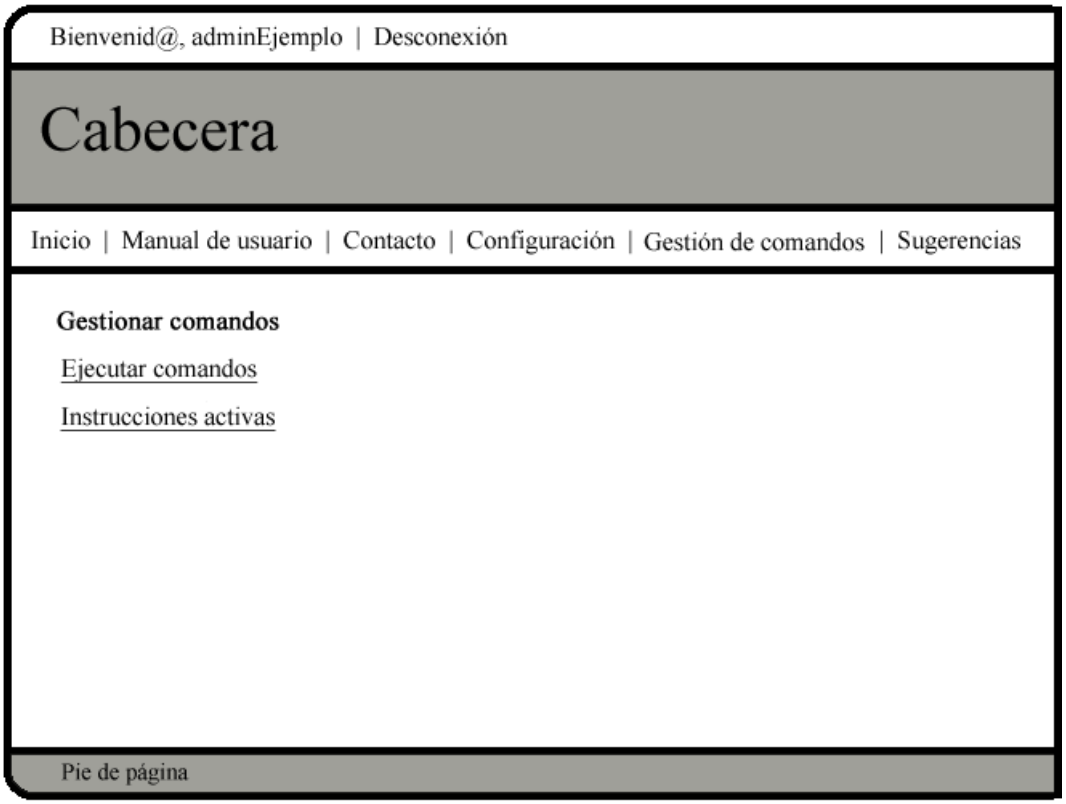


Figura 4.8: Diseño de la página del menú de gestión de comandos

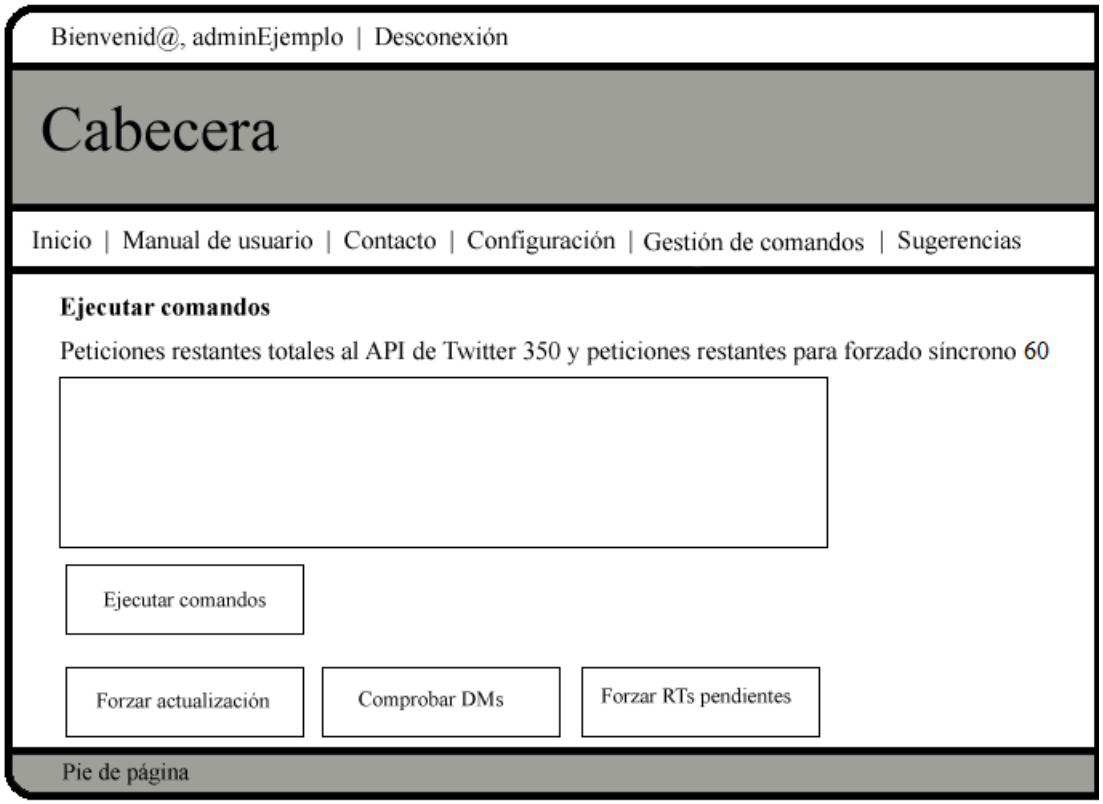


Figura 4.9: Diseño de la página de ejecutar comandos

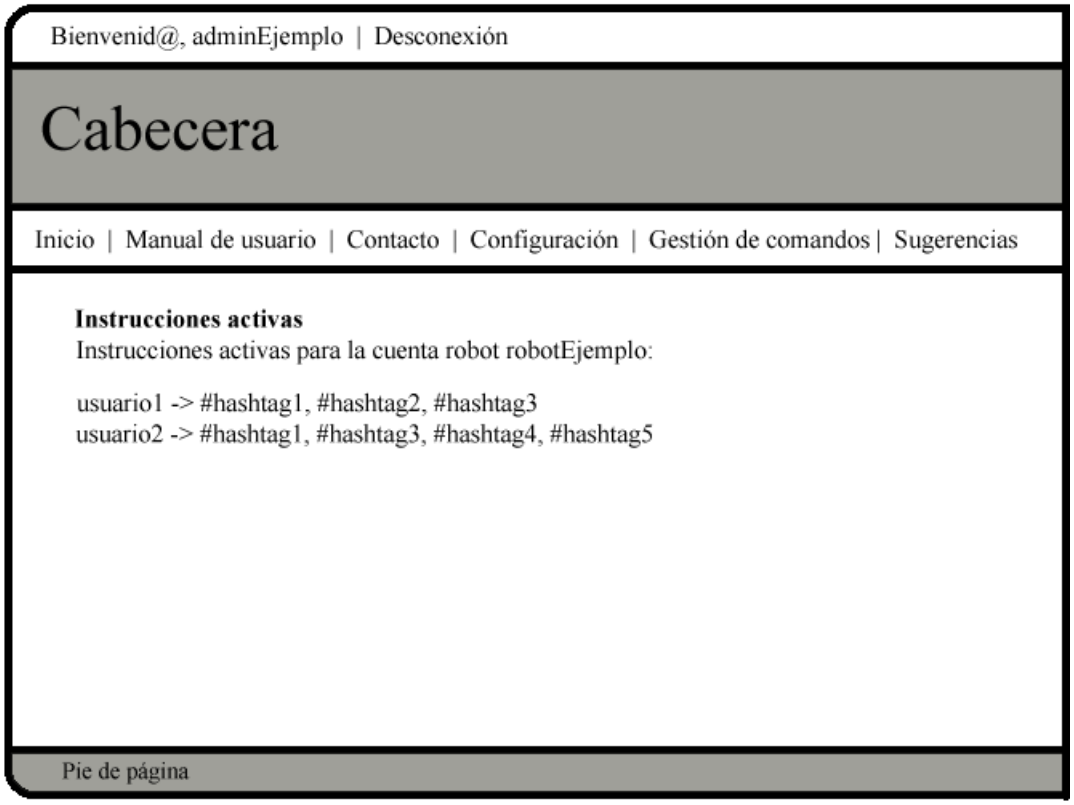


Figura 4.10: Diseño de la página de instrucciones activas

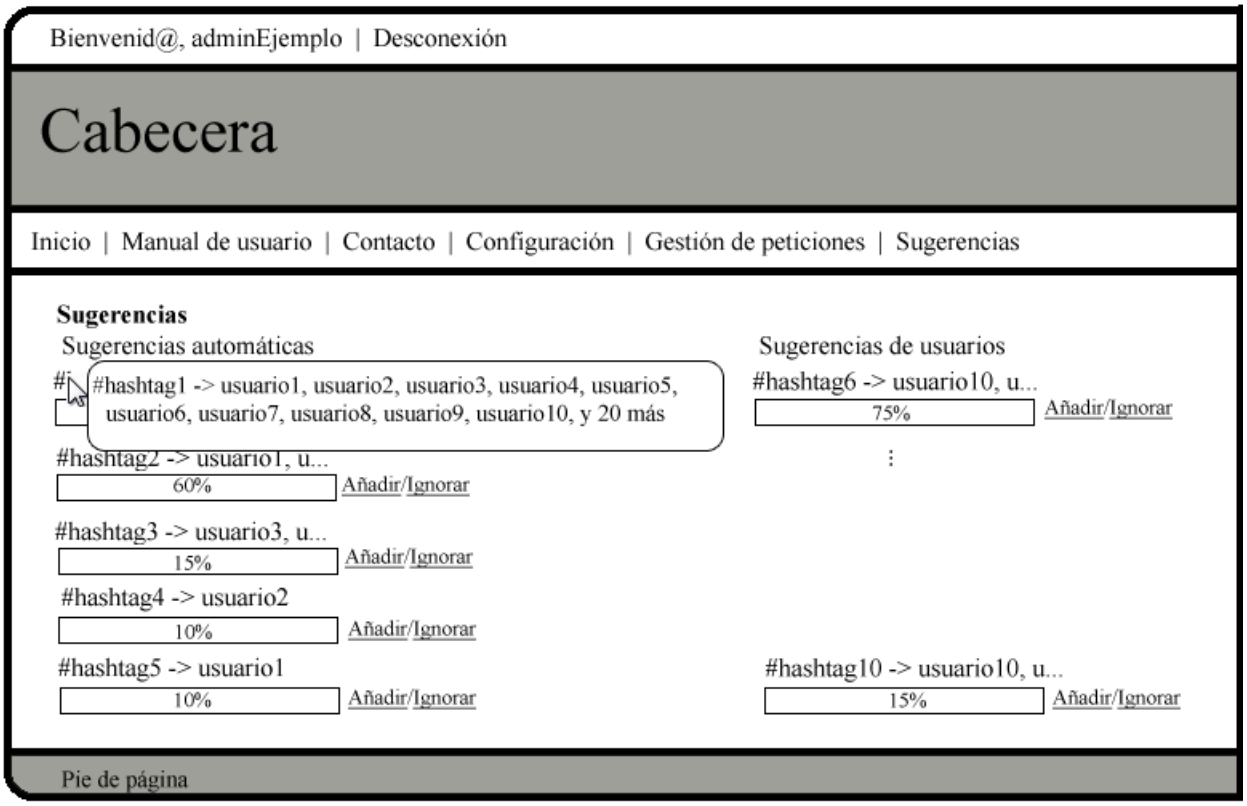


Figura 4.11: Diseño de la página de sugerencias

4.2.4. Pestaña de «Sugerencias»

Si el usuario pulsa sobre esta pestaña aparecerá una página en la que se podrán observar dos tipos de sugerencias de seguimiento (ver figura 4.11).

Por un lado se podrán ver las cinco sugerencias de seguimiento más relevantes que han sido calculadas internamente por el propio sistema. Cada una de estas sugerencias tendrá su correspondiente enlace de *«Añadir/Ignorar»* para que el administrador pueda añadirla a sus instrucciones de seguimiento o eliminarla si no le parece interesante. Si no hubiese ninguna sugerencia de este tipo, en vez de mostrarse la lista, se indicará al usuario un mensaje de texto informativo del tipo *«En este momento no hay sugerencias del sistema que mostrar»*.

Por otro lado, se mostrarán en una segunda columna las sugerencias de seguimiento enviadas por usuarios agregados a la cuenta robot. Al igual que ocurre con las sugerencias del sistema se visualizarán como máximo las cinco primeras. Si no hay, se indicará mediante un mensaje. También se podrán agregar o ignorar (ver figura 4.11).

Si se recomienda seguir un *hashtag* para muchos usuarios, al pasar el ratón por encima del texto incompleto se mostrará un *tooltip* con la sugerencia completa o, en el caso de que haya muchos usuarios, se indicará el nombre de diez usuarios seguido de la cantidad de usuarios que faltan por mostrar (ver figura 4.11).

Capítulo 5

Implementación y diseño del sistema

En este capítulo se hablará de la implementación y del diseño del sistema. Se especificará la estructura de ficheros de la aplicación, el diseño de la base de datos y la arquitectura.

Previamente a la implementación se instaló todo el *software* requerido para la fase de desarrollo y se dio de alta a la aplicación en *Twitter* desde la URL `https://dev.twitter.com/apps/new`. Con ello se obtuvieron la *consumerKey* y la *consumerSecret* que identifican a la aplicación de manera única. Para más información sobre la configuración leer el apéndice A.

5.1. Arquitectura del sistema

Como se ha mencionado en el capítulo 2 del presente documento, para la elaboración del código fuente se ha utilizado *Grails*, que sigue el patrón de diseño MVC (Modelo-Vista-Controlador) (ver figura 5.1).

La aplicación separará sus datos de la interfaz de usuario y de la lógica de negocio.

1. Modelo de datos: estará compuesto por las clases de dominio necesarias para modelar todos los datos que requieren almacenamiento persistente en el sistema. Para gestionar la base de datos se ha elegido *MySQL*.
2. Vistas: a través de ellas el usuario administrador podrá interactuar con el sistema para darse de alta y configurar su cuenta robot asociada. Estas vistas han sido elaboradas a través de GSPs y hojas de estilo CSS.
3. Controladores: son los encargados de recibir las órdenes que el usuario administrador ejecute a través de sus vistas correspondientes. También gestionarán toda la lógica de negocio del sistema. De esta parte se encargarán las clases controladoras y los servicios *Grails*.

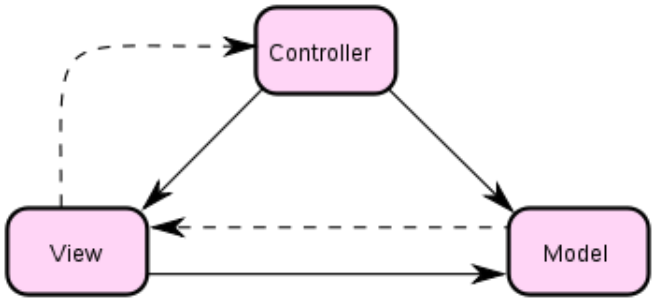


Figura 5.1: Esquema del MVC

5.2. Proceso de actualización de una cuenta robot

Antes de explicar la estructura del proyecto es importante definir qué operaciones debe hacer una cuenta robot para actualizarse.

1. Comprobar DMs. Se debe comprobar periódicamente si la cuenta robot ha recibido nuevos DMs. Si es este el caso, una vez recuperados, habrá que analizar si responden a algún comando de los permitidos y si han sido enviados desde la cuenta de *Twitter* del administrador (en caso de que se trate de comandos `ADDs` o `DELETEs`) o desde la cuenta de los usuarios (si se trata de comandos `SUGGESTIONs`). Si se comprueba que los comandos son correctos, se parsean y se ejecutan. Como se mencionó anteriormente, para el comando `UPDATE` no se permite la ejecución a través de DMs.

2. Comprobar si cada uno de los usuarios a los que sigue una cuenta robot ha publicado nuevos *tweets*. Si es así, para cada *tweet* hay que comprobar si debe ser *retwitteado* por la cuenta robot (esto es, si contiene *hashtags* marcados como relevantes para el usuario que lo haya publicado). En ese caso, guardar el *tweet* en la tabla de RTs pendientes.

3. Hacer RT de todos los *tweets* relevantes que hayan sido almacenados en la tabla de RTs pendientes.

Por otro lado, hay que tener en cuenta que el número de peticiones totales que *Twitter* permite hacer a la hora a su API REST es **limitado**, 350 si éstas están autenticadas. En este sistema las peticiones se envían siempre con *token* de autenticación *OAuth* para que su límite se mida en función al usuario de *Twitter* que las manda. En caso contrario, el límite descendería a 150 a la hora y se mediría por la IP desde la que las peticiones son enviadas.

En la aplicación las peticiones se pueden consumir tanto por la ejecución de tareas asíncronas automáticas como por peticiones síncronas enviadas por el administrador a través de la interfaz web. Por este motivo de las 350 se reservan **60 peticiones/hora para ejecuciones de tipo síncrono**.

El que el número de peticiones/hora a la API de *Twitter* sea limitado plantea un problema, sobre todo teniendo en cuenta la cantidad de peticiones que se pueden consumir al realizar las tres tareas de actualización que requiere una cuenta robot.

- Para obtener los DMs se debe hacer una petición HTTP a la API de *Twitter* al recurso REST `GET direct_messages`. En esta petición se especifica un parámetro con el *timestamp* del último DM más reciente leído. De este modo, solo se devolverán DMs posteriores a los obtenidos en anteriores comprobaciones.

  Es importante tener en cuenta que cada una de estas peticiones obtiene un máximo de 200 DMs, por ello se realiza paginación. En caso de que desde la última comprobación se hubiesen recibido más de 200 DMs en la cuenta robot, habría que hacer varias peticiones sucesivas hasta recuperarlos todos. Por ejemplo, si desde el último DM que se leyó, la cuenta robot hubiese recibido 210 más, se requeriría hacer dos peticiones seguidas a este recurso para poder recuperarlos todos (200 DMs * 1 petición + 10 DMs * 1 petición).

  El mínimo de peticiones que una cuenta robot gastaría en esta operación es de 1, y el número máximo depende del número de DMs recibidos desde la última comprobación.

- Por otro lado, habrá que hacer peticiones a `GET statuses/user_timeline` para recuperar los nuevos *tweets* publicados por cada usuario añadido a la cuenta robot. En este caso ocurre algo similar que lo explicado con los DMs. Cuando se hace una petición de este tipo para recuperar los

Nº usuarios cuenta robot	Nº usuarios máximos comprobados por intervalo	Espacio entre intervalos (minutos)
(0-25]	25	60
(25-50]	25	30
>50	30	10

Figura 5.2: Multiplexación de peticiones en función al número de usuarios de una cuenta robot

tweets de un usuario concreto, cada una permite obtener como máximo 200. Si hay más, hay que hacer paginación mediante peticiones sucesivas. Además, la API permite recuperar un máximo de 3200 *tweets* en total. Si algún usuario hubiese publicado más de 3200 *tweets* desde la última comprobación, el resto se perderían.

En la aplicación se ha decidido reducir el número de *tweets* obtenidos por página a 100 en vez de a 200. A priori puede parecer que esta decisión obliga a duplicar las peticiones en caso de que haya muchos *tweets* nuevos, pero tras realizar sucesivas pruebas se ha comprobado que la API REST de *Twitter* devuelve códigos de error en muchas ocasiones si se recuperan los *tweets* en grupos de 200. Al recogerse en grupos de 100, si un usuario hubiese publicado 550 *tweets* desde la última vez que se comprobó su *timeline*, para recuperarlos todos habría que hacer 6 peticiones (5 peticiones * 100 *tweets* + 1 petición * 50 *tweets*).

El mínimo de peticiones que una cuenta robot gastaría en esta operación es de 1 petición * nº usuarios, y el número máximo 32 peticiones * nº usuarios (ya que 32 peticiones * 100 *tweets* = 3200, que es el máximo número de *tweets* recuperables por usuario).

- Por último están las peticiones al recurso `POST statuses/retweet/:id` para hacer RT. En esta operación se consume una petición por cada RT.

  El mínimo de peticiones que una cuenta robot gastaría en esta operación es de 0 peticiones (si no hay RTs pendientes) y el número máximo no se puede conocer, depende del número de RTs pendientes que haya almacenados.

Lo que se ha decidido para solventar el problema con la restricción de peticiones es poner prioridades a cada una de las operaciones que hay que hacer para actualizar una cuenta robot. La operación más prioritaria será la 2 (comprobar *timelines* de los usuarios), y la menos prioritaria la 1 (comprobar DMs).

- Cada 6 horas se comprobará si se han recibido nuevos DMs. Si el administrador desea forzar esta comprobación lo podrá hacer desde su interfaz web por petición síncrona.
- Se calcularán los intervalos de comprobación en función al número de usuarios agregados a la cuenta robot. Para que no haya saturación de peticiones a la API, en caso de que una cuenta robot siga a muchos usuarios, éstos se irán comprobando en pequeños grupos, tal como se muestra en la tabla 5.2.
- Cuando todos los usuarios de una cuenta se hayan comprobado, se procederá a hacer los RTs pendientes. En caso de no haber suficientes peticiones para hacer los RTs, éstos serán *retwitteados* más tarde (o el administrador podrá forzarlo mediante petición síncrona desde su interfaz).

Por ejemplo, para una cuenta robot que tenga 100 usuarios agregados (`user1`, ..., `user100`):

1. Al tener más de 50 usuarios, según la tabla 5.2, cada 10 minutos se comprobarán grupos de 30 usuarios como máximo (si es que hay suficientes peticiones como para comprobar los 30). Si se supone que quedan 280 peticiones asíncronas libres en ese instante, y ninguno de los 30 primeros usuarios ha publicado más de 100 *tweets* desde la última comprobación:

   - Se accede a los *timelines* de los 30 primeros usuarios (del `user1` al `user30`). Si no se producen errores en las respuestas a las peticiones en ninguno, todos ellos se marcan como chequeados para esa cuenta robot (se pone su *flag* a 1). Las peticiones consumidas en este momento serán 30 (quedarían 280 - 30 = 250 peticiones libres).
   - Para cada uno de estos usuarios, de entre sus *tweets* nuevos obtenidos, se comprueba si hay relevantes. Si es así, se almacenan en una tabla de RTs pendientes.
   - Pasados otros 10 minutos, se vuelve a comprobar otro grupo de usuarios (del `user31` al `user60`). Suponer que el `user35` ha publicado 150 *tweets* desde la última vez que se chequeó su *timeline* y el resto no han publicado *tweets* nuevos o han publicado menos de 100. En este caso se consumirían 31 peticiones más a la API y quedarían libres 250 - 31 = 219. Si no se han producido errores en ninguna respuesta HTTP, se marca el *flag* a 1 de cada uno de estos usuarios.
   - Se comprueba si de entre todos los nuevos *tweets* obtenidos hay relevantes. Si es así, se almacenan en la tabla de RTs pendientes.
   - Pasados otros 10 minutos, se comprueba el tercer grupo de usuarios (del `user61` al `user90`). Suponer que hay tres usuarios de este grupo que han publicado cerca de 300 *tweets* desde la última vez que esa cuenta robot obtuvo sus *timelines*. El resto de usuarios no ha publicado nada o ha publicado menos de 100 *tweets*. En este caso, se necesitarían 27 usuarios * 1 petición + 3 usuarios * 3 peticiones = 36 peticiones. En este momento quedarían libres 219 - 36 = 183 peticiones. Los *flags* de cada uno de estos usuarios se ponen a 1.
   - Se almacenan todos los *tweets* relevantes obtenidos en la tabla de RTs pendientes.
   - Pasados otros 10 minutos, se comprueba el cuarto grupo de usuarios (del `user91` al `user100`). En este caso suponer que ninguno ha publicado más de 100 *tweets* desde el último chequeo a sus *timelines*. Se requerirían entonces 10 peticiones, y en total quedarían libres 183 - 10 = 173. Se marcan todos los *flags* correspondientes a 1. En este momento se habría terminado de comprobar a todos los usuarios de la cuenta robot.
   - Se almacenan en la tabla de RTs pendientes todos los *tweets* relevantes obtenidos para el cuarto grupo de usuarios.

2. Como los *flags* de todos los usuarios están a 1, se vuelven a poner a 0 para siguientes comprobaciones, y se procede a hacer los RTs pendientes. Suponer que hay almacenados en la tabla 20 *tweets* que *retwittear*. Al haber suficientes peticiones, la cuenta robot hace los 20 RTs. Si todos se realizan de manera correcta, entonces se eliminan de la tabla de pendientes. Si alguno fallase, no se eliminaría y se intentaría reenviar más tarde.

   En el caso de que no hubiese suficientes peticiones para hacer todos los RTs, éstos se intentarían hacer al final de la siguiente tanda de comprobaciones. Suponiendo que el caso sea muy extremo y a lo largo del día siempre quedasen RTs pendientes, se reservará un período que va de las 3:00 a las 4:00 de la mañana para *retwittearlos*.

   En el período de las 3:00 a las 4:00 de la mañana no se chequearán cuentas de usuarios para dejar peticiones libres para posibles RTs pendientes.

5.3. Cálculo de sugerencias

La frecuencia de aparición de las sugerencias determina la relevancia que tiene cada una de ellas. Dicha frecuencia se calcula en base a las veces que aparece un *hashtag* concreto, o bien en los *tweets* de los usuarios a los que sigue una cuenta robot (en el caso de las auto-sugerencias), o bien en los comandos de tipo `SUGGESTION` (en el caso de las sugerencias manuales).

Para que un *hashtag* se considere sugerencia, no debe estar ya siguiéndose para el usuario para el que se contabiliza.

Para entender este mecanismo, a continuación se detalla su funcionamiento mediante ejemplos.

5.3.1. Cálculo de auto-sugerencias

Suponer que uno de los usuarios a los que sigue una cuenta robot, el `usuario1`, publica los siguientes *tweets*:

1. "`Hoy se registrarán temperaturas de 30° a la sombra #ElTiempo #ElTiempo`"
2. "`@usuario3, he leído que hoy habrá temperaturas de hasta 30° a la sombra #ElTiempo`"
3. "`@usuario2, hoy habrá temperaturas de hasta 30° a la sombra`"

Suponer que el *hashtag* `#ElTiempo` no se sigue para ningún usuario de la cuenta robot.

Del primer *tweet* se registrará el *hashtag* `#ElTiempo`, como dicho *hashtag* ha sido publicado por el `usuario1`, en ese momento la sugerencia será:

`#ElTiempo -> usuario1`, con una frecuencia de aparición 2 (puesto que el *hashtag* aparece por duplicado)

Tras la publicación y el análisis del segundo *tweet*, al haber sido publicado por el `usuario1` y al haber una mención en dicho *tweet*, el *hashtag* se contabilizará una vez por cada mención más la vez referente al usuario que ha publicado el *tweet* en cuestión. Al tratarse del mismo *hashtag* del anterior *tweet*, y estar la sugerencia almacenada ya, ésta modificará su frecuencia e incluirá a los usuarios que corresponda. En este caso la sugerencia quedará:

`#ElTiempo -> usuario1, usuario3` con una frecuencia de aparición 2 (acumulados del anterior *tweet*) + 2 ((1 mención + 1 usuario creador del *tweet*) * 1 aparición del *hashtag*) = 4

Por último, tras publicarse y analizarse el tercer *tweet*, se observa que aparece una nueva mención pero ningún *hashtag*, por lo tanto, este *tweet* no se considerará como sugerencia. Las menciones se consideran únicamente cuando en el mismo *tweet* hay un *hashtag* asociado (o varios). Esto se debe a que todas las sugerencias, para ser válidas, deben incluir un *hashtag* y un usuario como mínimo.

5.3.2. Cálculo de sugerencias manuales

En el caso de las sugerencias de los usuarios, se contabilizarán las sugerencias siguiendo un patrón similar al anterior.

Por ejemplo, si un usuario envía las siguientes sugerencias:

```
SUGGESTION -u usuario1, usuario2, usuario3 -q #Meteosat, #meteosat, #Verano,
#Alergia, #Primavera
SUGGESTION -u usuario1 -q #Meteosat, #Verano
```

Las sugerencias serán las siguientes (teniendo en cuenta que, en este caso, en una misma `SUGGESTION` se eliminan *hashtags* y menciones duplicadas):

`#Meteosat -> usuario1, usuario2, usuario3`, con una frecuencia de aparición de 4 (3 usuarios * 1 *hashtag* del tipo `#Meteosat` + 1 usuario * 1 *hashtag* del tipo `#Meteosat`)

`#Verano -> usuario1, usuario2, usuario3`, con una frecuencia de aparición de 4 (3 usuarios * 1 *hashtag* del tipo `#Verano` + 1 usuario * 1 *hashtag* del tipo `#Verano`)

`#Alergia -> usuario1, usuario2, usuario3`, con una frecuencia de aparición de 3 (3 usuario * 1 *hashtag* del tipo `#Alergia`)

`#Primavera -> usuario1, usuario2, usuario3`, con una frecuencia de aparición de 3 (3 usuario * 1 *hashtag* del tipo `#Primavera`)

5.4. Estructura de la aplicación

En la figura 5.3 se puede observar la jerarquía de paquetes de la aplicación, que es la estructura básica que sigue cualquier proyecto *Grails*.

5.4.1. Directorio grails-app

Dentro del directorio `grails-app` se incluyen todos los componentes de la aplicación: controladores, servicios, clases de dominio, ficheros de mensajes para la internacionalización, etc.

Conf

La carpeta `grails-app/conf` incluye por defecto los ficheros necesarios para la configuración de la aplicación, tal y como se explicó anteriormente en el apartado 2.2.13 del capítulo 2.

En esta aplicación, el contenido de este directorio es el que se muestra en la figura 5.4.

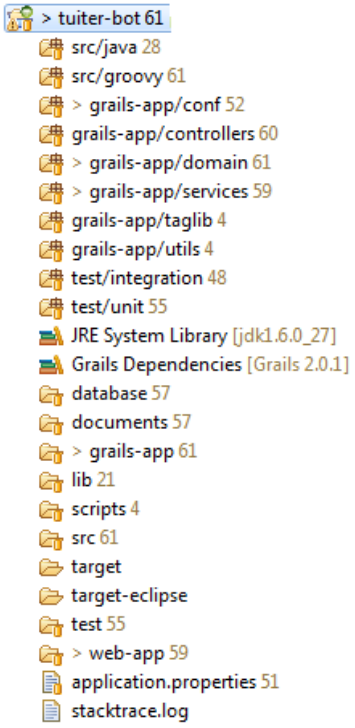


Figura 5.3: Estructura de paquetes del proyecto

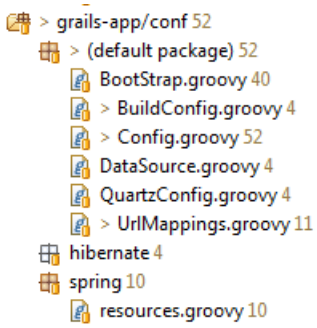


Figura 5.4: Subdirectorios y ficheros contenidos en grails-app/conf

Todos los ficheros y subdirectorios que se pueden ver se generan automáticamente al crear el proyecto en *Grails*, excepto `QuartzConfig.groovy`, que se crea al instalar el plugin de *Quartz*.

A continuación se indica qué ficheros han sido modificados con respecto a su configuración por defecto.

- `Config.groovy`. En este fichero se ha modificado la configuración que viene por defecto de los mensajes de *log*. Es necesario que éstos se muestren en todos los componentes de la aplicación que deban proporcionar información al usuario para tareas de depuración. Además, se han añadido todas las variables globales que la aplicación utiliza. A continuación se muestra el fragmento que hace referencia a la configuración de las variables globales:

```
...
// twitter application properties
twitter {
```

```
    consumerKey = "**********"
    consumerSecret = "****************************************"
    callbackUrl =
    "http://localhost:8080/tuiter-bot/accountManager/callback"
}

// mail properties
mail {
    mailSender = "tuiter.bot"
    passMailSender = "******"
}

// remainingHits values properties
requestLimits {
    totalByHour = 350
    asyncReq = 290
    syncReq = 60
}
```

Cada una de estas variables indica:

- `consumerKey` y `consumerSecret`: cadenas de caracteres alfanuméricos generadas al dar de alta a la aplicación en *Twitter*.
- `callbackUrl`: URL de *callback* a la que será redirigido el administrador tras dar permisos desde una página segura de *Twitter* para que la aplicación acceda a los datos de su cuenta robot (DMs, *timeline*, ...). La URL que por defecto se indica supone que la aplicación está ejecutándose en el servidor local (*localhost*), en caso de que se instale en otro servidor externo habrá que cambiar el servidor y el puerto indicados en esta dirección.
- `mailSender`: nombre de usuario de la cuenta de correo electrónico de *Gmail* desde la que la aplicación envía los correos de sugerencias.
- `passMailSender`: contraseña de la dirección de correo electrónico indicada en la propiedad anterior.
- `totalByHour`: este parámetro indica el número de peticiones totales que *Twitter* permite hacer a la hora a su API REST.
- `asyncReq`: número máximo de peticiones/hora (a la API REST de *Twitter*) de tipo asíncrono que se reservan por cuenta robot.
- `syncReq`: indica el número máximo de peticiones/hora (a la API REST de *Twitter*) de tipo síncrono que se reservan por cuenta robot.

Para acceder a las variables globales desde cualquier componente de la aplicación, habrá que inyectar una dependencia al objeto `grailsApplication`:

```
class AccountManagerService {
    ...
    def grailsApplication
    ...
    }
```

Una vez inyectada la dependencia al objeto, si desde el servicio anterior se quisiese llamar a la variable `consumerKey`, habría que escribir la línea:

```
grailsApplication.config.twitter.consumerKey
```

- `DataSource.groovy`. En este archivo se indican los datos correspondientes a la conexión con la base de datos *MySQL* de la aplicación (nombre de usuario, contraseña, URL).

```
dataSource {
        pooled = true
        driverClassName = "com.mysql.jdbc.Driver"
        dialect = "org.hibernate.dialect.MySQL5InnoDBDialect"
        username = "root"
        password = "root"
        dbCreate = "update"
        url = "jdbc:mysql://localhost:3306/tb"
}
```

Controllers

En el directorio `grails-app/controllers` se encuentran los controladores *Grails*. Esta aplicación consta únicamente de un controlador denominado `AccountManagerController.groovy`, que se encargará de ejecutar las acciones que un administrador elija a través de las vistas y de decidir la siguiente vista que le debe mostrar (ver figura 5.5).

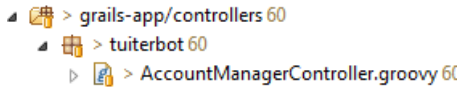


Figura 5.5: Subdirectorios y ficheros contenidos en grails-app/controllers

Para que el controlador disponga de un código más limpio, en vez de implementar directamente la lógica de negocio, llama a métodos de negocio creados en los servicios. También gestionará posibles errores con el fin de mostrar mensajes de advertencia a través de la interfaz web (formularios rellenos con datos incorrectos, campos vacíos, etc.).

A continuación se muestra un fragmento del controlador que contiene el código necesario para que un administrador inicie sesión en el sistema:

```
class AccountManagerController {
    // inyección del servicio gestor de la cuenta robot
    def accountManagerService
    // inyección del servicio gestor del administrador
    def adminManagerService
    // inyección del servicio encargado de hacer peticiones a la API de Twitter
    def checkerService

    def grailsApplication
    def messageSource // inyección del messageSource

    /**
     * Accion a la que redirige el formulario de autenticacion
```

```
     * de un administrador. Autentica un administrador en el
     * sistema. Si entra por vez primera, se le redirige a la pag.
     * de Twitter en la que se configura la cuenta robot y se dan
     * permisos para que la aplicacion pueda acceder a ella mediante
     * la Twitter API
     */
    def loginAsAdmin = {

        def user = session.getAttribute("user")
        def role = session.getAttribute("role")
        if (user != null && role != null) {
            redirect action: index
            return false
        }
        else {

            // si hay algún campo en blanco. ERROR.
            if(params.screenName == "" || params.password == ""){
                flash.message = messageSource.getMessage('fields.msg',
                null, null)
            }
            else {
                def admin = Admin.findByScreenName(params.screenName)
                // si el administrador no está registrado, ERROR.
                if (admin == null) {
                    flash.message = messageSource.getMessage('userm.msg',
                    null, null)
                }
                else {
                    if(admin.password == params.password.encodeAsMD5()){
                        admin.password = " "
                        session.setAttribute("user", admin)
                        session.setAttribute("role", "admin")
                        Account acc = Account.findByAdmin(admin)
                        if (acc != null) {
                            redirect action:index
                            return true
                        }
                        else {
                            try {
                                redirect url: accountManagerService.authenticate()
                            } catch (TwitterException te) {
                                flash.message = messageSource.getMessage(
                                'te.msg', [
                                    te.getStatusCode(),
                                    te.getErrorMessage()
                                ].toArray(), null)
                                log.info flash.message
                                redirect action: index
                                return false
                            }
                            return true
                        }
                    }
                    else {
                    // si password no coincide. ERROR.
```

```
                        flash.message = messageSource.getMessage(
                        'password.msg', null, null)
                    }
                }
            }
            redirect action: loginAdmin
            return false
        }
    }

    /**
     * Formulario de autenticacion del administrador.
     * Redirige a la pag. de autenticacion del administrador
     * (loginAdmin.gsp).
     */
    def loginAdmin = {
        def user = session.getAttribute("user")
        def role = session.getAttribute("role")
        if (user != null && role != null) {
            redirect action: index
            return false
        }
        return true
    }
        ...
    }
```

Se puede observar que el fragmento de código mostrado consta de dos acciones, `loginAsAdmin` y `loginAdmin`. La acción `loginAdmin` se encarga de redirigir al usuario a la vista que contiene el formulario de autenticación. La otra acción, `loginAsAdmin`, se ejecuta cuando el administrador envía los datos de autenticación desde el formulario, y su función es comprobar si estos datos son o no correctos. En el código de esta acción, lo primero que se hace es acceder a la base de datos y comprobar si el nombre de usuario que se ha indicado coincide con el de algún administrador registrado en el sistema. Si no es así, se redirige al usuario a la misma página (el formulario de autenticación) con `redirect action: loginAdmin` y se le muestra un mensaje indicando que los datos introducidos son incorrectos. En caso de que se encuentre un administrador con el nombre indicado pero su contraseña no coincida con la que esté almacenada en base de datos (guardada en *md5*), también se mostrará al usuario el correspondiente mensaje de error.

Si la autenticación resulta exitosa, se comprobará si ese administrador tiene ya asociada una cuenta robot. Si es así, se guardarán los datos del usuario en sesión, redirigiéndole a su página de inicio (mediante `redirect action: index`). Si no es así, se redirigirá al administrador a la URL de *callback* que devuelve el método `authenticate` del servicio `AccountManagerService`. Este método se encargará de llevar a cabo la autenticación *OAuth* de la cuenta robot para conseguir sus *tokens* de acceso y poder acceder a sus datos en *Twitter* (DMs, *timeline*, etc.).

Si se produce algún error en este proceso de conseguir los *tokens*, se lanzará una `TwitterException` que se capturará en el código, mostrándose un mensaje de error al usuario en la interfaz.

Cuando se llama a `redirect action: loginAdmin` o a `redirect action: index`, se está redirigiendo al usuario a la vista `loginAdmin.gsp` o a `index.gsp`, respectivamente.

Services

Dentro de este directorio, `grails-app/services`, se encuentran los servicios que definen la lógica de negocio de la aplicación (ver figura 5.6). Se han implementado tres servicios diferentes. Dos de ellos son llamados solamente por el controlador, y el tercero es llamado tanto por la clase *Quartz* que ejecuta tareas automáticas como por el controlador.

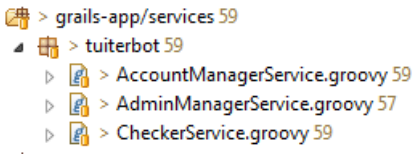


Figura 5.6: Subdirectorios y ficheros contenidos en grails-app/services

- `AccountManagerService.groovy`: es el servicio encargado de realizar las funciones necesarias para asociar un administrador con una cuenta robot concreta. Para ello, define los métodos para conseguir los *tokens* de autenticación de una cuenta robot (con *OAuth*).
- `AdminManagerService.groovy`: es el servicio que gestiona algunas de las operaciones que se realizan desde la interfaz web del administrador (obtener la lista de instrucciones asociadas a una cuenta robot, obtener las sugerencias y auto-sugerencias de relevancia, y registrar a un administrador en el sistema).
- `CheckerService.groovy`: contiene métodos para actualizar las cuentas robots que estén registradas en el sistema. Es llamado tanto por la clase que ejecuta tareas automáticas (para actualizar una cuenta robot asíncronamente) como por el controlador (peticiones síncronas ejecutadas por el administrador desde la interfaz web).

A continuación se muestra un fragmento de código del servicio `AccountManagerService`. En él se obtienen los *tokens* de autenticación que posibilitan el acceso a la información en *Twitter* de una cuenta robot concreta:

```
class AccountManagerService {

    static transactional = true
    static scope = 'session'

    TwitterUtil tu
    def grailsApplication
    def messageSource

    /**
     * Metodo para autenticacion OAuth de una cuenta de Twitter
     * a traves de la callbackUrl
     *
     * @return url de autorizacion
     * @throws TwitterException, si se produce error al acceder a la
     * API de Twitter
     */
    String authenticate() throws TwitterException {
        tu = new TwitterUtil(
```

```
                grailsApplication.config.twitter.consumerKey,
                grailsApplication.config.twitter.consumerSecret,
                grailsApplication.config.requestLimits.asyncReq,
                grailsApplication.config.requestLimits.syncReq)
        return tu.authenticate(grailsApplication.config.twitter.callbackUrl)
    }

    /**
     * Metodo para autenticacion OAuth (verificar credenciales)
     *
     * @param oauth_verifier
     * @param admin, administrador de la cuenta robot de la que se
     * verifican credenciales
     * @return cuenta autenticada con sus correspondientes tokens
     * de acceso o null si error
     * @throws TwitterException
     */
    Account verifyCredentials(String oauth_verifier, Admin admin)
    throws TwitterException {
        User aux = tu.verifyCredentials(oauth_verifier)
        log.info aux
        // se busca si ya hay alguna cuenta asociada al administrador
        Account accountByAdmin = Account.findByAdmin(admin)
        /* se busca si ya esta registrada para otro administrador
         * la cuenta robot que se desea asignar
         */
        Account accountFind = Account.findByTwitterId(aux.id)
        if (accountFind != null &&
        accountFind.admin.screenName != admin.screenName) {
            log.info "Se intenta asignar a otro admin una cuenta robot asignada."
            return null
        }
        //se persiste la cuenta en BD
        //(si no está ya, y si no, se actualizan sus valores)
        if (accountByAdmin == null) {
            def timestamp = TwitterUtil.getTimestamp()
            accountByAdmin = new Account(twitterId: aux.id,
                screenName: aux.screenName,
                    profileImg: aux.profileImageURL.toString(),
                    oauthToken: tu.accessToken.getToken(),
                    oauthTokenSecret: tu.accessToken.getTokenSecret(),
                    batchTime: 60, lastSinceDateDM: timestamp,
                    admin: admin)
        }
        else {
            // si se renueva la cuenta robot asociada al administrador
            // se elimina la configuración ->usuarios, instrucciones
            // activas, RTs pendientes, sugerencias
            // y autosugerencias
            accountByAdmin.delete(flush:true)
            def timestamp = TwitterUtil.getTimestamp()
            accountByAdmin = new Account(twitterId: aux.id,
                    screenName: aux.screenName,
                    profileImg: aux.profileImageURL.toString(),
                    oauthToken: tu.accessToken.getToken(),
                    oauthTokenSecret: tu.accessToken.getTokenSecret(),
```

```
                    batchTime: 60, lastSinceDateDM: timestamp,
                    admin: admin)
        }
        if (!accountByAdmin.validate()) {
            return null
        }
        return accountByAdmin.save(flush:true)
    }

...
}
```

Domain

`grails-app/domain` contiene las clases de dominio necesarias para definir el modelo de datos de la aplicación (ver figura 5.7). En este caso se han modelado las siguientes clases:

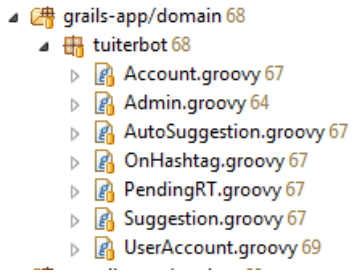


Figura 5.7: Subdirectorios y ficheros contenidos en grails-app/domain

- `Account.groovy`. Representa a una cuenta robot, y contiene todos los atributos necesarios para la configuración de la misma.
- `Admin.groovy`. Representa al administrador de una cuenta robot. La asociación entre el administrador y su cuenta robot es 1:1.
- `Autosuggestion.groovy`. Esta entidad representa una sugerencia de seguimiento que ha sido calculada automáticamente por el sistema.
- `Suggestion.groovy`. Representa una sugerencia de seguimiento que ha sido enviada a la cuenta robot por uno de sus usuarios agregados.
- `OnHashtag.groovy`. Representa un *hashtag* activo para un usuario concreto de una cuenta robot. De esta entidad se extraen las instrucciones activas para una cuenta robot.
- `PendingRT.groovy`. Representa a un *tweet* relevante para una cuenta robot y pendiente de ser *retwitteado*.
- `UserAccount.groovy`. Esta entidad indica los nombres de los usuarios de *Twitter* que pertenecen a una cuenta robot determinada (usuarios agregados por cuenta robot).

A continuación se muestra como ejemplo el código de la clase de dominio que representa a una cuenta robot.

```
package tuiterbot

/**
 * Clase de dominio que representa una cuenta robot
 *
 * @author Marta
 * @version 1.0
 */
class Account {

    // identificador numérico de usuario cuando se conecta con Twitter
    Long twitterId
    // nombre de usuario único de Twitter
    String screenName
    // url de la imagen de perfil
    String profileImg
    // token de la cuenta
    String oauthToken
    // token secreto de la cuenta
    String oauthTokenSecret
    // administrador de la cuenta robot
    Admin admin
    // identificador de último DM leído de una cuenta
    Long lastIdDM
    // timestamp del último DM leído de la cuenta robot
    Long lastSinceDateDM
    // tiempo de chequeo modo batch
    int batchTime

    static hasMany = [users: UserAccount, suggestions: Suggestion,
    autosuggestions: AutoSuggestion]

    static constraints = {
        twitterId (nullable: false, unique: true)
        screenName (nullable: false, blank: false,
        matches: /^[a-zA-Z0-9_]{1,15}$/, unique: true)
        profileImg (nullable: false, blank: false)
        oauthToken (nullable: false, blank: false, unique: true)
        oauthTokenSecret (nullable: false, blank: false, unique: true)
        admin (nullable: false, blank: false,
        matches: /^[a-zA-Z0-9_]{1,15}$/, unique: true)
        lastIdDM (nullable: true)
        lastSinceDateDM (nullable: false)
        batchTime (nullable: false)
    }

    static mapping = {
        table "account"
        twitterId column: "twitter_id"
        screenName column: "screen_name"
        profileImg column: "profile_img"
        oauthToken column: "oauth_token"
        oauthTokenSecret column: "oauth_token_secret"
        admin column: "admin"
        lastIdDM column: "last_iddm"
```

```
        lastSinceDateDM column: "last_since_datedm"
        batchTime column: "batch_time"
        version false
    }

    String toString() {
        "Account with [twitterId: ${twitterId}, screenName: ${screenName}] "
    }

}
```

Tal y como se puede observar, se declaran todos los atributos necesarios para mapear la entidad que representa a una cuenta robot. En el bloque `constraints` se definen las restricciones que se deben validar antes de persistir un objeto de este tipo (el formato de los nombres de usuario, las claves únicas, etc.), y el bloque `mapping` da nombre a cada una de las columnas de la tabla y a la propia tabla.

Views

Todas las vistas de la aplicación se han diseñado respetando las reglas que se definieron en el capítulo 4 del presente documento.

Están contenidas en el directorio `grails-app/views`, y se han implementado mediante GSPs y hojas de estilo CSS (ver figura 5.8).

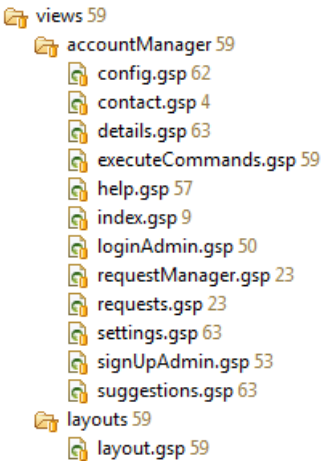


Figura 5.8: Subdirectorios y ficheros contenidos en grails-app/views

Dentro de este directorio se encuentran dos carpetas más:

- `grails-app/views/layouts`. *Grails* permite incluir archivos denominados *layouts* dentro de este directorio, en ellos se define la estructura común que tendrán las vistas. En la aplicación se ha creado una plantilla denominada `layout.gsp` que define la estructura común de todas las vistas (referencias a las hojas de estilo utilizadas, menú de inicio de sesión, cabecera, menú de opciones, contenido de la vista, y pie de página).
- `grails-app/views/accountManager`. Contiene las vistas de la aplicación.
    - `index.gsp`. Página de inicio de la aplicación (ver figura 5.9).

Figura 5.9: Inicio

- `contact.gsp`. Vista de información de contacto (ver figura 5.10).
- `help.gsp`. Vista que contiene un manual de ayuda al usuario con instrucciones sobre cómo utilizar la aplicación (ver figura 5.11).
- `signUpAdmin.gsp`. Vista a través de la que un administrador se puede registrar en el sistema (ver figura 5.12).
- `loginAdmin.gsp`. Vista que contiene el formulario de autenticación del administración en el sistema (ver figura 5.13).
- `config.gsp`. Vista del administrador para acceder al menú de configuración de la cuenta robot (permite acceder a `settings.gsp` o a `details.gsp`) (ver figura 5.14).
- `details.gsp`. Vista del administrador que contiene los detalles de configuración de su cuenta robot (usuarios agregados, administrador asociado y tiempo entre comprobaciones) (ver figura 5.15).
- `settings.gsp`. Vista del administrador que contiene un formulario para introducir la información necesaria para la configuración de una cuenta robot (ver figura 5.16).
- `requestManager.gsp`. Vista de menú de comandos (permite acceder a `requests.gsp` o a `executeCommands.gsp`) (ver figura 5.17).
- `executeCommands.gsp`. Vista de ejecución de comandos del administrador (ver figura 5.18).
- `requests.gsp`. Vista que muestra al administrador las instrucciones de seguimiento activas para su cuenta robot (ver figura 5.19).
- `suggestions.gsp`. Vista de sugerencias a la que solo puede acceder un administrador. Muestra las cinco sugerencias más prioritarias enviadas por usuarios de la cuenta robot (si

Figura 5.10: Información de contacto

las hay), y las cinco sugerencias más prioritarias calculadas de manera automática por el sistema (si las hay). Permite agregarlas a las instrucciones de seguimiento o ignorarlas si no resultan de interés (ver figura 5.20).

Jobs

El directorio `grails-app/jobs` contiene una clase denominada `TaskReminderJob.groovy` (ver figura 5.21). Esta clase se encarga del envío de correos electrónicos con sugerencias de seguimiento y de la ejecución automática de todas las tareas de actualización de las cuentas robots dadas de alta en el sistema (peticiones asíncronas).

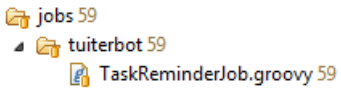


Figura 5.21: Subdirectorios y ficheros contenidos en grails-app/jobs

Los jobs se han programado mediante expresiones *cron* y disparadores sencillos (*simple triggers*), que indican la hora a la que se tiene que ejecutar cada uno.

En el código fuente de la clase `TaskReminderJob.groovy` se debe inyectar una dependencia al servicio `CheckerService` para poder llamar a los métodos que se encargan de actualizar las cuentas robots dadas de alta en el sistema. Por un lado, se deben programar las expresiones necesarias (*cron* o disparadores) para ejecutar cada una de las tareas en tiempos determinados. Estas expresiones se definen en

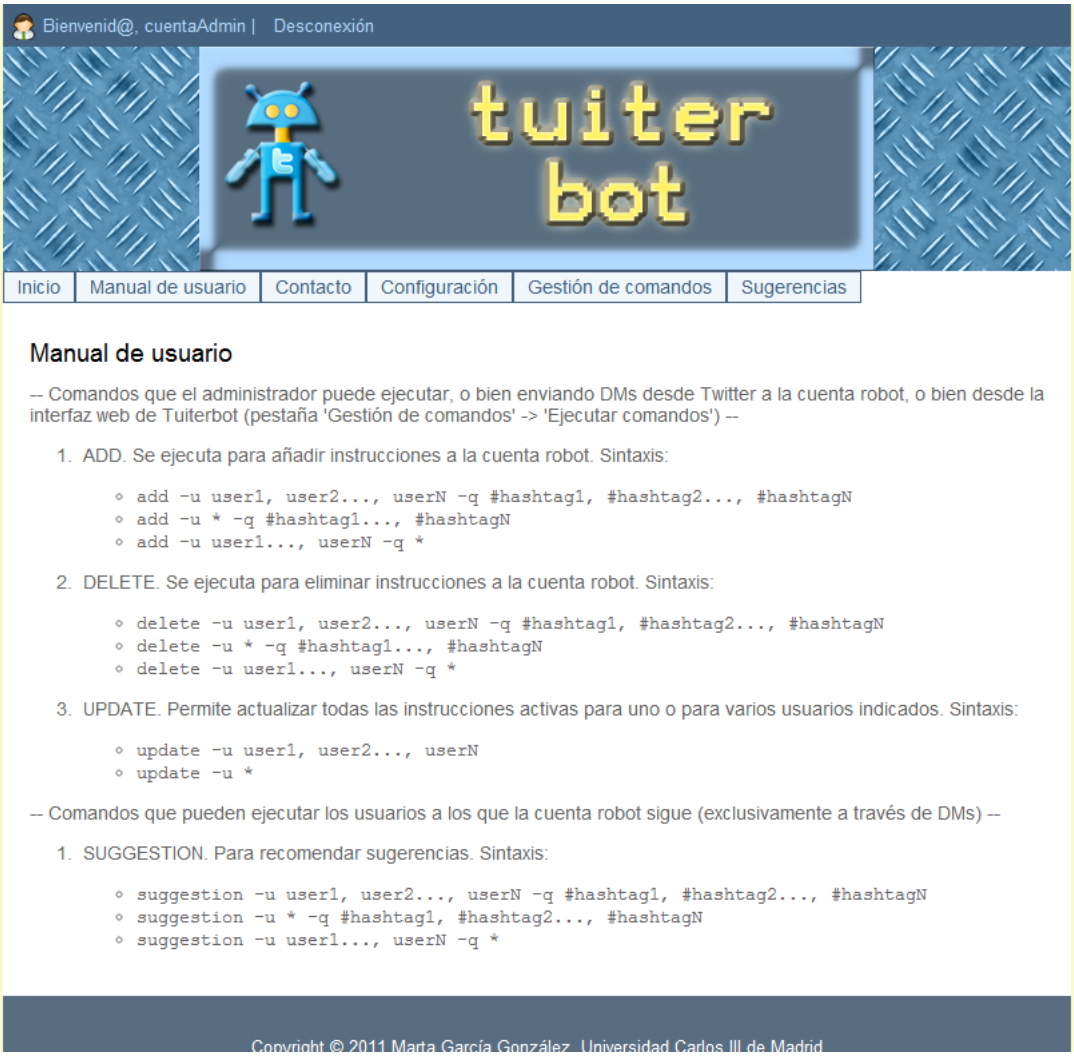


Figura 5.11: Manual de usuario

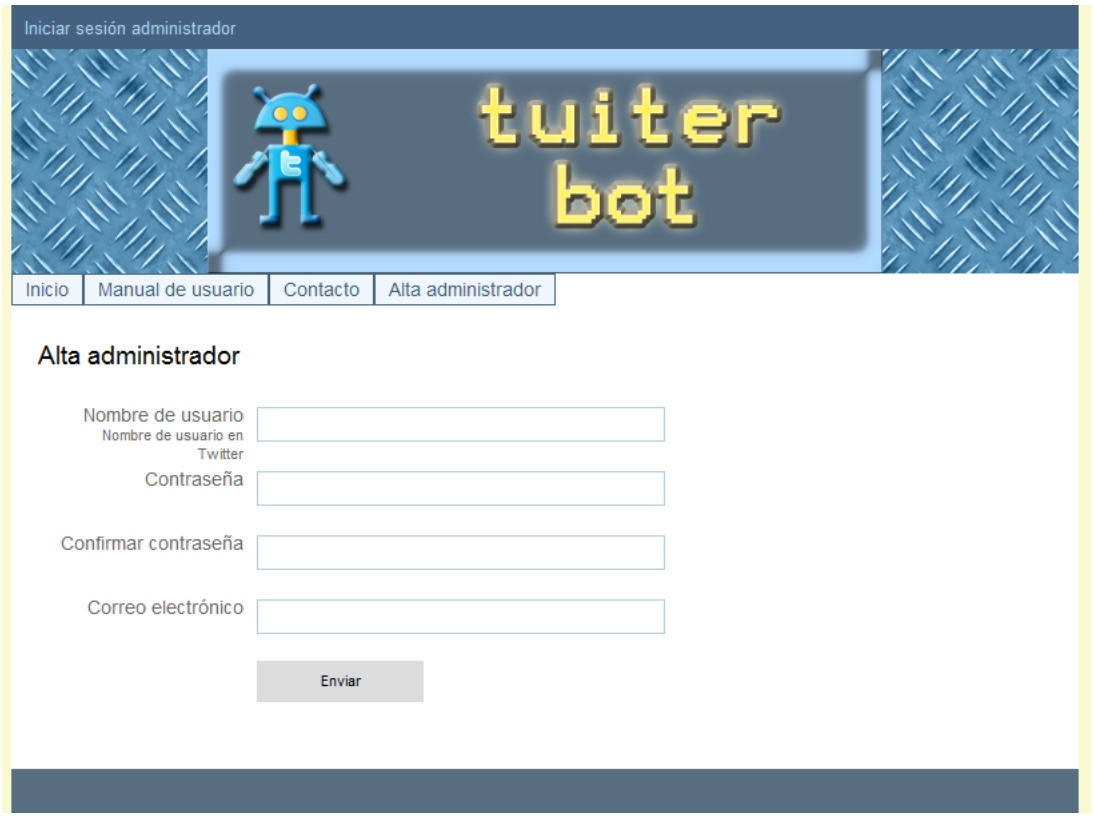


Figura 5.12: Inicio sesión de administrador

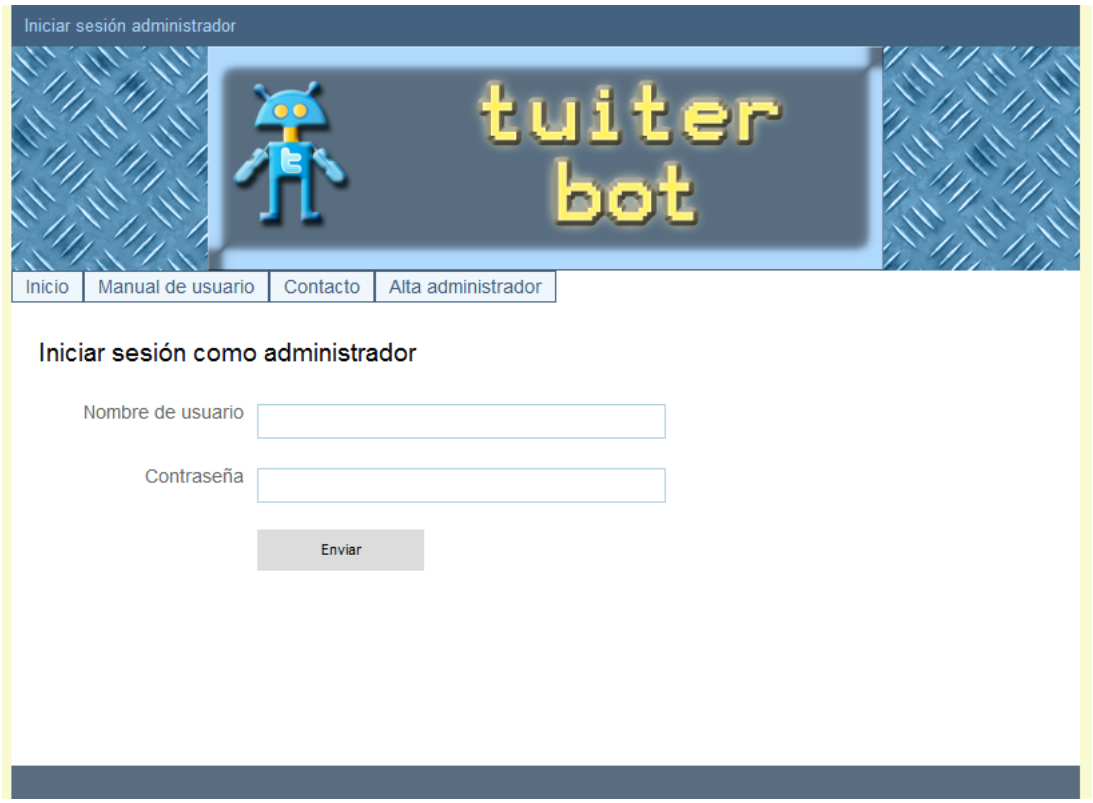


Figura 5.13: Vista de inicio de sesión

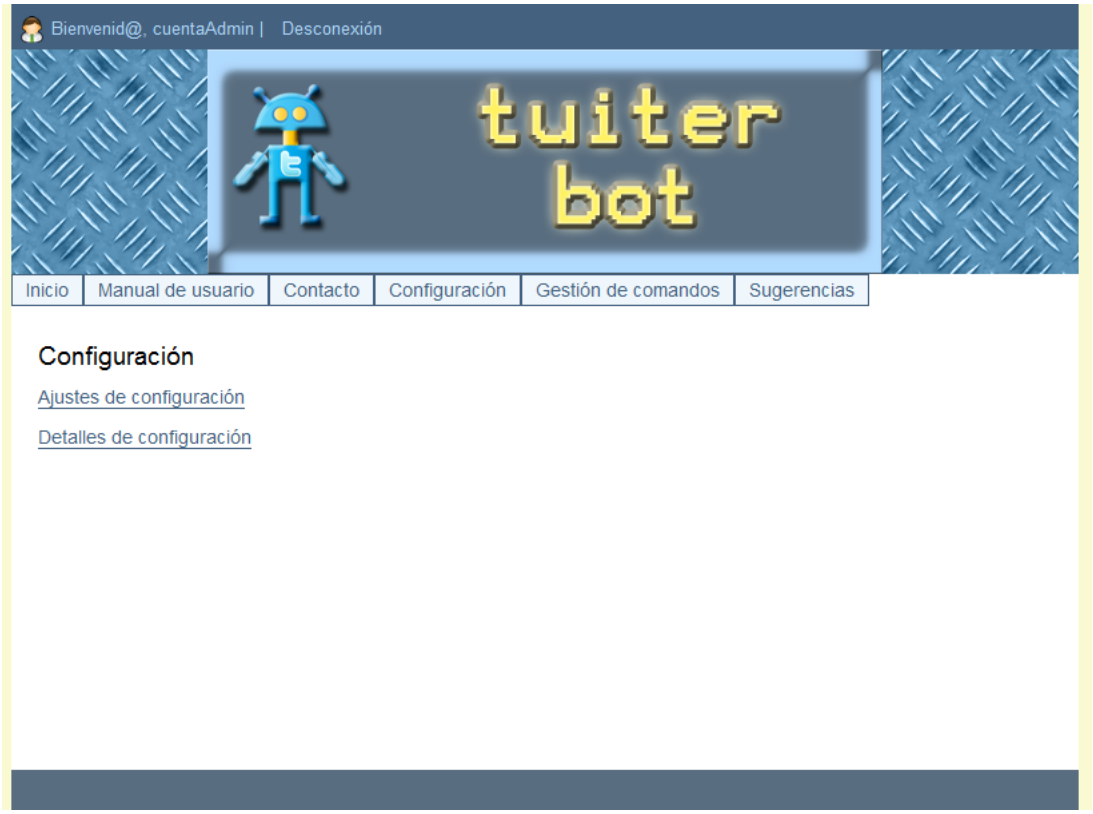


Figura 5.14: Menú de configuración

Figura 5.15: Vista de detalles

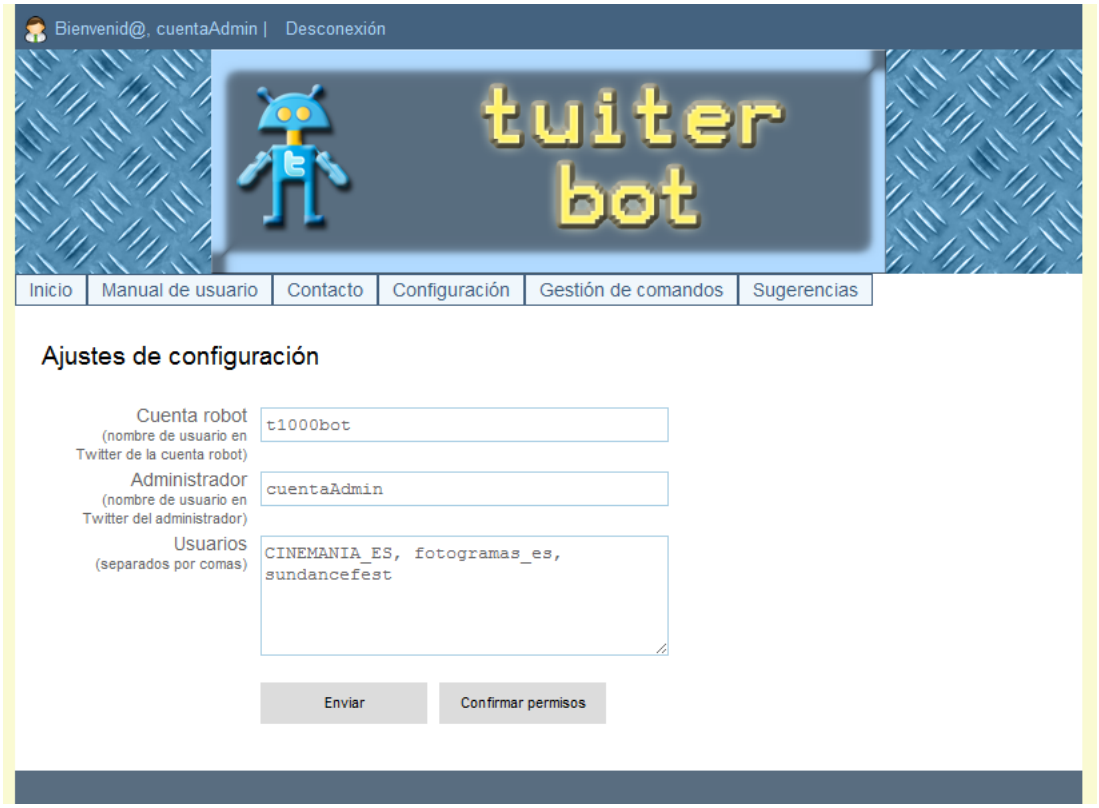


Figura 5.16: Formulario de configuración

Figura 5.17: Menú de gestión de comandos

Figura 5.18: Vista de ejecución de comandos

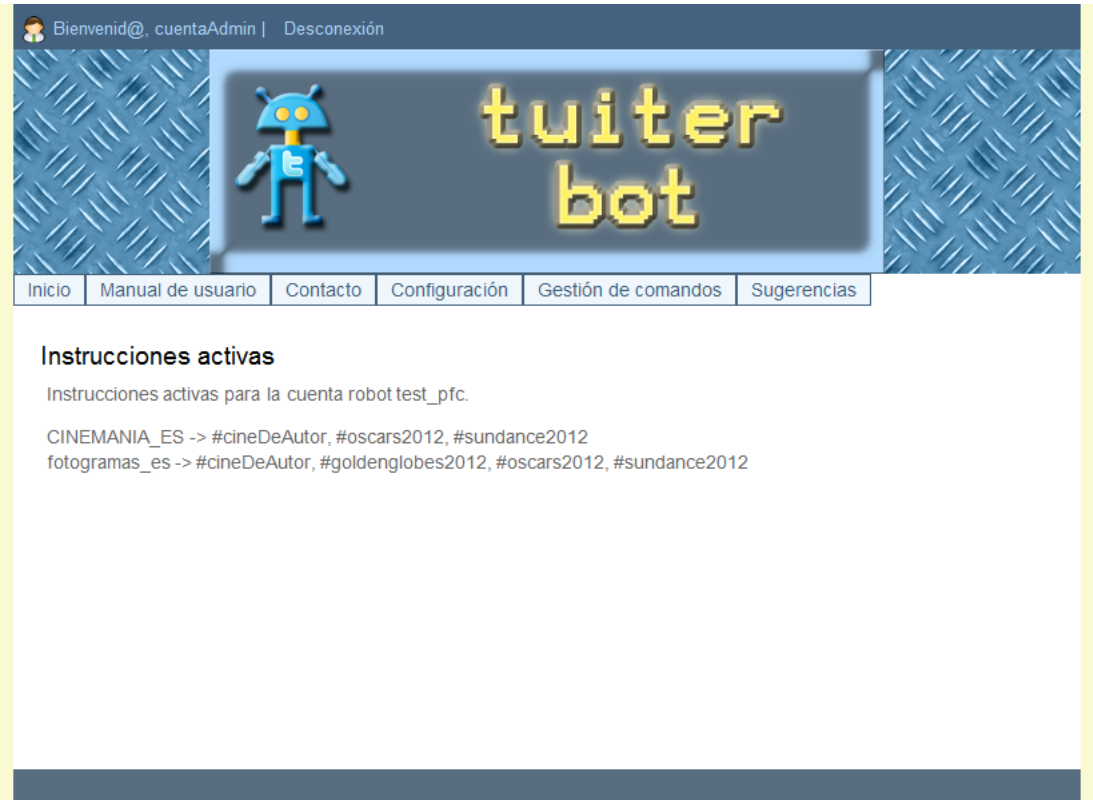


Figura 5.19: Vista de instrucciones de seguimiento

el interior de `static triggers`, y a cada una se le debe indicar los instantes en los que se tiene que lanzar y un nombre que la identifique.

Por otro lado, se define el método `def execute(context)`, que es llamado cada vez que se activa un disparador. Dentro de este método habrá que identificar, con el contexto que se pasa como parámetro, cuál de las acciones es la que se ha lanzado. En función de eso se llamará a un método u a otro del servicio inyectado.

A continuación se muestran fragmentos de código de esta clase. únicamente aparecen las tareas programadas relacionadas con el envío de correos de sugerencias y con la comprobación de DMs:

```
class TaskReminderJob {

    def checkerService

    static triggers = {
    ...
        // a las 12:00 del mediodía correos de sugerencias
        cron name: 'suggestionsMail', cronExpression: "0 00 12 ? * *"
        // cada 6 horas comprueba DMs
        simple name: 'checksDMs', startDelay: 60000, repeatInterval: 21600000
        ...
    }

    def execute(context) {
```

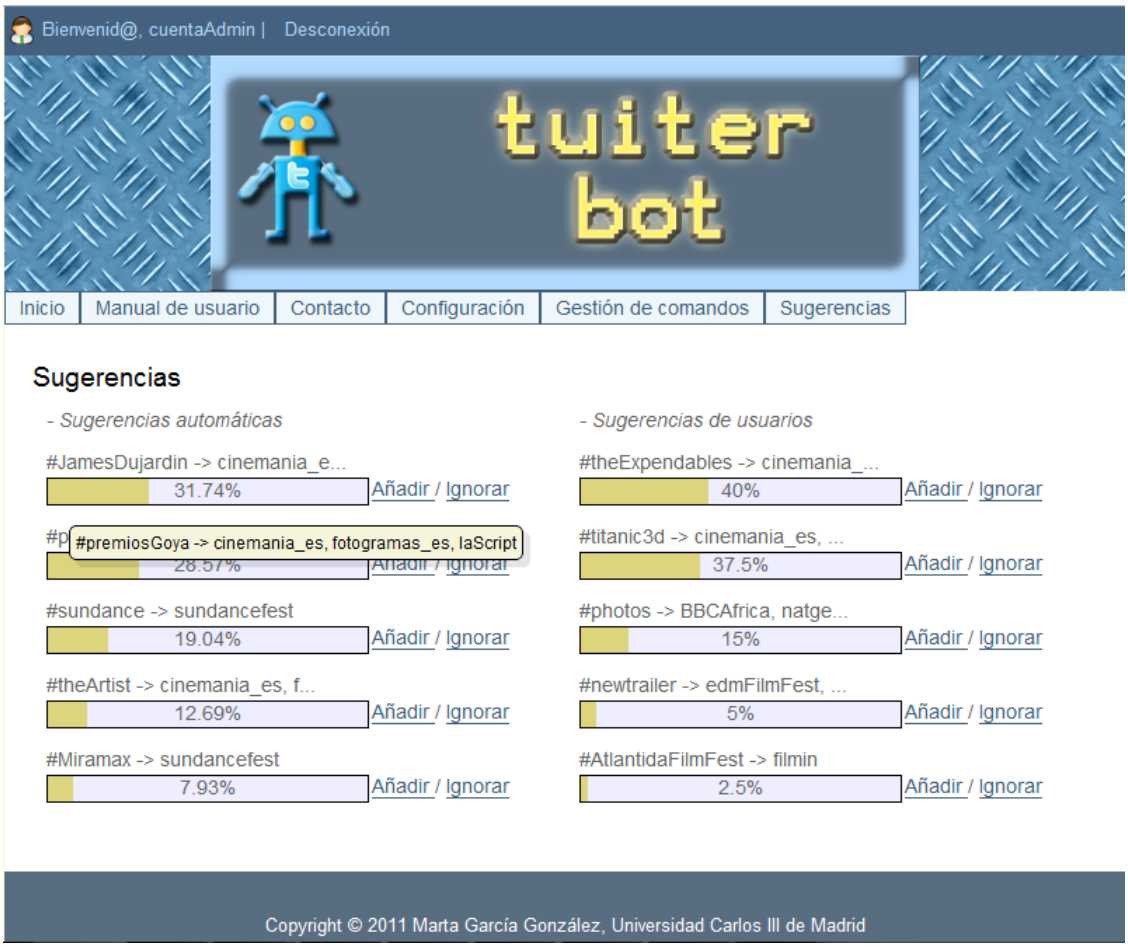


Figura 5.20: Vista de sugerencias

```
        def triggerName = context.getTrigger().getKey()

        if (triggerName.toString().equals("GRAILS_JOBS.checksDMs")) {
            //cada 6 horas comprueba si hay nuevos DMs para cada cuenta
            // y si los hay, los parsea
            checkerService.checkDMsForAccounts()
        }
        else if (triggerName.toString().equals(
        "GRAILS_JOBS.suggestionsMail")) {
            // envío de mails a los administradores con sugerencias.
            // todos los días a las 12:00 AM
            checkerService.sendMailSuggestions()
        }
        log.info(triggerName)
        log.info("Task reminders sent on ${new Date()}")
    }
    ...
}
```

i18n

En *Grails* se tiene soporte para la internacionalización por medio de ficheros de recursos almacenados en la carpeta `grails-app/i18n`. Estos archivos de recursos son ficheros `.properties` de *Java* en los que se deben definir las distintas versiones de los mensajes que contendrá la aplicación en todos los idiomas que soporte.

En este caso, se han personalizado los mensajes para el idioma inglés y el castellano a través de los ficheros:

- `messages_properties`: por defecto en castellano.
- `messages_en.properties`: para el idioma inglés.
- `messages_es.properties`: para el idioma castellano.

Grails sabrá qué idioma escoger en función de la cabecera `Accept-Languaje` que todos los navegadores envían en cada petición.

Para usar los mensajes que se definen en estos archivos desde una GSP, hay que utilizar la etiqueta `message`. Por ejemplo:

En un fragmento de la GSP del formulario de autenticación de un administrador en el sistema, se usa:

```
<g:message code="screenName.label" />
```

De este modo, se busca un mensaje definido en el archivo de propiedades correspondiente y cuyo código es `screenName.label`.

```
screenName.label = Nombre de usuario
```

5.4.2. Directorio src

Dentro del directorio `src` hay dos subdirectorios, `src/java` y `src/groovy`.

Java

Como se ha comentado en capítulos anteriores, cualquier proyecto *Grails* es compatible con código *Java*. En el caso de la aplicación desarrollada se ha reutilizado una clase `.java` que envía peticiones HTTP y cuyo nombre es `Http.java`. Se usará para averiguar el *timestamp* del servidor de *Twitter* mediante peticiones HTTP al recurso `https://api.twitter.com/1/help/test.json` de su API REST, consultando la cabecera `Date` que se devuelve.

Esta petición se utiliza en dos situaciones:

1. Cuando un administrador da de alta una cuenta robot en el sistema. De este modo se sabrá en qué momento ha sido dada de alta la cuenta robot. La primera vez que se acceda a los DMs recibidos en dicha cuenta, se obtendrán únicamente los que se hayan recibido en instantes posteriores al momento en el que se dio de alta a la cuenta robot.

2. Cuando se activa una nueva instrucción de seguimiento mediante el comando `ADD`. Por ejemplo, si un administrador ejecuta el comando `ADD -u usuario1 -q #hashtag1`, el *hashtag* `#hashtag1` será relevante únicamente en *tweets* que publique el usuario `usuario1` en instantes posteriores a la ejecución del comando `ADD`.

Groovy

Dentro de `src/groovy` irán los ficheros `.groovy` utilizados en el código que no sean ni controladores, ni clases de dominio, ni servicios. Fundamentalmente se incluirán clases que resulten de utilidad para el desarrollo de la aplicación. En el proyecto se han añadido tres ficheros de este tipo:

- `DMQueryParser.groovy`: esta clase contiene los métodos necesarios para la validación de los comandos que puede ejecutar un administrador (`ADD`, `DELETE` o `UPDATE`) o un usuario agregado a una cuenta robot (`SUGGESTION`). Estos comandos han sido definidos con una sintaxis específica y siempre que se ejecuten desde la aplicación (o a través de DMs) deben cumplir estrictamente este formato. En caso contrario se considerarán incorrectos.

- `MailUtil.groovy`: esta clase sirve para enviar correos electrónicos. En la aplicación se utiliza para enviar diariamente a un administrador los correos electrónicos de sugerencias de seguimiento.

- `TwitterUtil.groovy`: esta clase es una de las más importantes, ya que contiene todos los métodos necesarios para acceder a la API de *Twitter* haciendo uso de la librería *Twitter4j*. Contiene métodos para autenticar a un usuario de *Twitter* mediante *OAuth*, obtener los DMs de una cuenta de *Twitter*, hacer RT de un *tweet*, obtener los *tweets* de un usuario concreto, obtener el *timestamp* del servidor de *Twitter* (usando la clase `Http.java` mencionada previamente), y obtener el número de peticiones restantes a la API de *Twitter*.

  Esta clase también gestiona el número de peticiones que una cuenta robot concreta ha consumido a la API de *Twitter* (tanto síncronas como asíncronas).

5.4.3. Directorio lib

En el directorio lib es donde se incluyen todas las librerías requeridas. En el caso de esta aplicación se han añadido:

- `mysql-connector-java-5.1.7-bin.jar`: para la base de datos *MySQL*.

- `twitter4j-core-2.2.4.jar`: es una librería *Java* que posibilita hacer llamadas a la API de *Twitter*.
- `imap.jar`, `pop.jar`, `smtp.jar`, `dns.jar` y `mailapi.jar`: necesarias para la clase que envía correos electrónicos.

5.4.4. Directorio web-app

Es la raíz de la aplicación web y contiene todos los recursos web utilizados: archivos *Javascript*, hojas de estilo CSS e imágenes (ver figura 5.22).

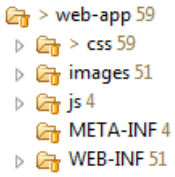


Figura 5.22: Subdirectorios y ficheros contenidos en web-app

En esta aplicación se han utilizado dos hojas de estilo CSS (`layout.css` y `css_style.css`) para conseguir la presentación requerida en las distintas pantallas de la interfaz web.

Como ejemplo, se muestra el fragmento de código CSS que da formato a los campos de texto de los formularios:

```
.textField {
        float: left;
        padding: 4px 2px;
        border: solid 1px #aacfe4;
        width: 300px;
        margin: 2px 0 20px 10px;
        font: 0.9em courier;
        color: #666;
}
```

Se establece el tamaño, el color y el tipo de fuente; el ancho de la caja de texto; la anchura del borde de la caja y su color; y los márgenes derecho, izquierdo, superior e inferior.

5.5. Diseño de la base de datos

Para su correcto funcionamiento, el sistema desarrollado requiere almacenar algunos datos de manera permanente. Para ello se ha diseñado una base de datos relacional que ha sido gestionada con *MySQL* a través de *Grails*.

5.5.1. Modelo Entidad-relación (E-R)

Como se puede observar en el modelo E-R (ver figura 5.23), hay siete entidades, aunque la entidad que representa a un usuario, al guardar únicamente el valor del nombre de usuario (*screen_name*), se ha suprimido para simplificar el diseño de la base de datos. Resultado de las asociaciones representadas en el modelo quedan las siguientes tablas:

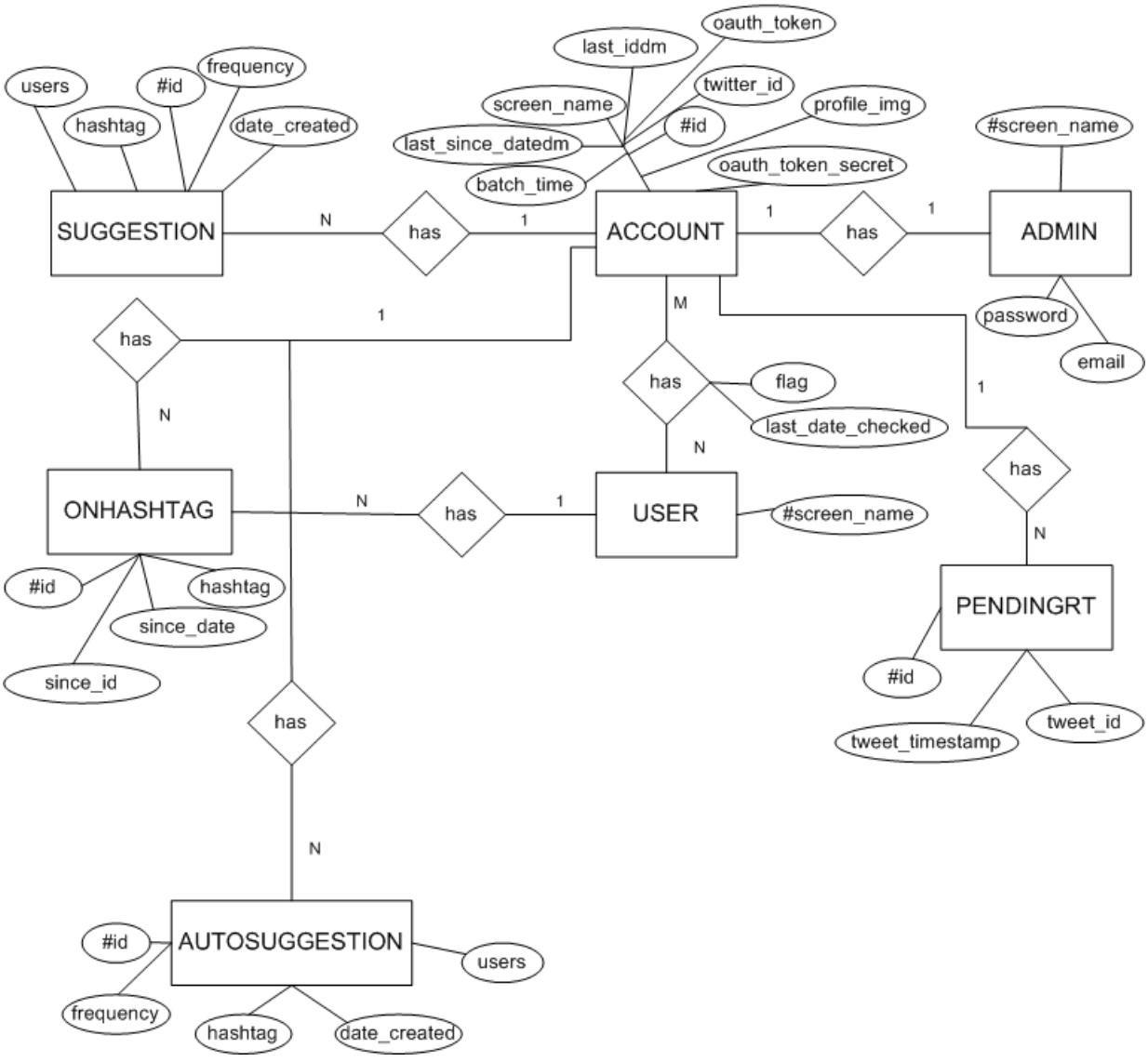


Figura 5.23: Modelo E-R de la aplicación

1. *Admin*: esta entidad representa a un administrador de una cuenta robot. Hay que recordar que puede haber un y solamente un administrador por cuenta robot. Un administrador no puede administrar más de una cuenta simultáneamente y, además, una cuenta robot no puede ser administrada por varios administradores a la vez.

    - *screen_name*: clave primaria y nombre de usuario de *Twitter* del administrador.
    - *password*: contraseña para entrar en el sistema (no tiene por qué coincidir con la contraseña de *Twitter*).
    - *mail*: cuenta de correo electrónico del administrador.

2. *Account*: esta entidad representa a la cuenta robot en sí. En ella se incluyen los datos de configuración de la misma y otros datos necesarios para el correcto funcionamiento de la aplicación.

    - *id*: clave primaria e identificador auto-incremental de la cuenta robot.
    - *twitter_id*: es clave única, este valor no se puede repetir para dos cuentas diferentes y es el número identificador único de la cuenta robot en *Twitter*.
    - *screen_name*: nombre de usuario de la cuenta robot en *Twitter*. Es clave única.
    - *profile_img*: URL de la dirección de la imagen de perfil de la cuenta robot en *Twitter*.
    - *oauth_token*: *token* para la autenticación *OAuth* de la cuenta robot. Es clave única.
    - *oauth_token_secret*: *token* secreto para la autenticación *OAuth* de la cuenta robot. Es clave única.
    - *admin*: administrador de la cuenta robot. Es clave única (ya que un administrador solamente puede administrar una cuenta robot).
    - *batch_time*: tiempo entre intervalos de comprobación de la cuenta robot. Se calcula dinámicamente, en función al número de usuarios agregados a la cuenta robot.
    - *last_iddm*: identificador del último DM leído de la cuenta robot.
    - *last_since_datedm*: *timestamp* del último DM leído de la cuenta robot.

3. *UserAccount*: esta entidad surge de la asociación N:M de la entidad que representa a un usuario y de la entidad que representa a una cuenta robot. Indica la relación entre un usuario y la cuenta robot a la que ha sido agregado.

    - *id*: identificador auto-incremental y clave primaria.
    - *account*: identificador de la cuenta robot a la que está agregado el usuario.
    - *user*: nombre del usuario de *Twitter* agregado a la cuenta robot.
    - *last_date_checked*: *timestamp* del último *tweet* del usuario *user* que se ha chequeado en busca de auto-sugerencias para la cuenta *account*.
    - *flag*: bandera que indica si el usuario *user* ha sido chequeado en busca de *tweets* relevantes para la cuenta robot en un intervalo determinado.
    - (*user*, *account*) son clave única conjunta. Estos dos valores no se pueden repetir simultáneamente en dos tuplas distintas de esta tabla.

4. *OnHashtag*: Los *onHashtags* son instrucciones de seguimiento que el administrador ha activado. Cada uno representa un *hashtag* relevante para un usuario concreto de una cuenta robot.

    - *id*: identificador auto-incremental único y clave primaria.
    - *user_account*: usuario asociado a una cuenta robot para el que ha sido activado el *onHahstag*.

- *hashtag*: *hashtag* marcado como relevante en publicaciones del usuario indicado.
- *since_id*: identificador del último *tweet* del usuario indicado que se comprobó para el *onHashtag*.
- *since_date*: *timestamp* del último *tweet* del usuario indicado que se comprobó para el *onHashtag*.
- (*user_account*, *hashtag*) son clave única conjunta.

5. *PendingRT*: en esta entidad se almacenan todos los *tweets* que cumplen alguna condición que este definida en los *onHashtags*. Una cuenta robot debe hacer RT de todos los *tweets* que estén asignados a ella en esta tabla. Se almacenan previamente por seguridad, para que no se pierdan. Hasta que no se garantice que los *tweets* pendientes han sido *retwitteados* por la cuenta robot, no serán eliminados de la tabla.

   - *id*: identificador auto-incremental único y clave primaria.
   - *tweet_id*: identificador único del *tweet* a *retwittear*.
   - *tweet_timestamp*: *timestamp* del *tweet* a *retwittear*. Si hay muchos *tweets* pendientes para una cuenta robot, éstos serán *retwitteados* en orden cronológico, del más antiguo al más nuevo.
   - *account*: cuenta robot que debe hacer el RT.
   - (*tweet_id*, *account*) son clave única conjunta.

6. *Suggestion*: las sugerencias son enviadas por los usuarios que pertenecen a la comunidad de una cuenta robot. Éstos pueden sugerir a la cuenta robot que siga *hashtags* para nuevos usuarios, o también, que siga algún *hashtag* nuevo para alguno de los usuarios pertenecientes a la cuenta robot.

   - *id*: identificador auto-incremental y clave primaria.
   - *account*: cuenta robot a la que se le propone la sugerencia.
   - *users*: usuarios a los que se sugiere separados por comas, no tienen que tener relación necesariamente con los usuarios agregados a la cuenta robot.
   - *hashtag*: *hashtag* al que se sugiere seguir para el conjunto de usuarios indicado en *users*.
   - *frequency*: frecuencia con la que aparece la sugerencia (si ha sido sugerida dos veces, este campo valdrá 2).
   - *date_updated*: fecha en la que se ha producido la última actualización en la frecuencia de la sugerencia.
   - (*account*, *hashtag*) son clave única conjunta.

7. *Autosuggestion*: las auto-sugerencias siguen el mismo formato que las sugerencias, pero en este caso no las envían los usuarios sino que son calculadas automáticamente por el sistema.

   - *id*: identidicador auto-incremental y clave primaria.
   - *users*: usuarios a los que se sugiere separados por comas, no tienen que tener relación necesariamente con los usuarios agregados a la cuenta robot.
   - *hashtag*: *hashtag* al que se sugiere seguir para el conjunto de usuarios indicado en *users*.
   - *frequency*: frecuencia con la que aparece la auto-sugerencia. Si el usuario indicado en *user_account* publica muchos *tweets* con el *hashtag* propuesto, cada vez que aparezca dicho *hashtag* en su *timeline* se incrementará este valor.

- *date_updated*: fecha en la que se ha producido la última actualización en la frecuencia de la auto-sugerencia.
- (*account*, *hashtag*) son clave única conjunta.

Capítulo 6

Pruebas

Al finalizar el desarrollo de la aplicación, ésta fue sometida a un conjunto de pruebas unitarias y funcionales para comprobar su correcto funcionamiento. El sistema debe cumplir todos los requisitos especificados en el capítulo 3 del presente proyecto.

A continuación se describen todas estas pruebas realizadas.

6.1. Pruebas unitarias

Todas las pruebas unitarias se alojan dentro del directorio raíz del proyecto, concretamente en el subdirectorio `test/unit`.

6.1.1. Pruebas unitarias de dominio

Para probar si las clases de dominio se han mapeado correctamente y se han definido bien las restricciones que son de esperar para cada entidad almacenada, se han realizado una serie de pruebas unitarias que chequean posibles errores de diseño en la base de datos.

Para ello, por cada clase de dominio existente en la aplicación se ha implementado una clase de test. Dicha clase hereda de `GrailsUnitTestCase`, que contiene métodos para que los objetos se comporten en pruebas como lo harían en ejecución.

En el caso de las pruebas de las clases de dominio se ha utilizado el método `mockForContraints`, que permite comprobar el funcionamiento de las reglas de validación en clases de entidad.

Un ejemplo de código de estas pruebas es:

```
package tuiterbot

import grails.test.*

class AccountTests extends GrailsUnitTestCase {

    void testConstraints() {
        // mock administrador
        def existingAdmin = new Admin(screenName: "adminAccount",
                password: "jjjjj", mail: "mail@gmail.com")
        mockForConstraintsTests(Admin, [existingAdmin])
```

```
        // users, lastIddm pueden ser valores nulos inicialmente
        // se crea un mock de la cuenta robot
        def existingAccount = new Account(id: 0, screenName: "cuentaprueba",
                oauthToken: "esteesunoauthtokendeprueba",
                oauthTokenSecret: "esteesunoauthtokensecretdeprueba",
                profileImg: "urlImg" ,
                twitterId: 111111L,
                batchTime: 15,
                admin: existingAdmin,
                lastSinceDateDM:1328207838749)

        mockForConstraintsTests(Account, [existingAccount])

        // Ahora se prueba a introducir otra cuenta robot los
        // valores que deben ser unicos, duplicados (screenName, admin,
        // oauthToken, oauthTokenSecret, profileImg y twitterId)
        def account = new Account(id: 0, screenName: "cuentaprueba",
                oauthToken: "esteesunoauthtokendeprueba",
                oauthTokenSecret: "esteesunoauthtokensecretdeprueba",
                profileImg: "urlImg" ,
                twitterId: 111111L,
                batchTime: 15,
                admin: existingAdmin,
                lastSinceDateDM:1328207838749)
        // se comrpueba que da error donde se espera
        assertFalse account.validate()
        assertEquals "unique", account.errors["screenName"]
        assertEquals "unique", account.errors["oauthToken"]
        assertEquals "unique", account.errors["oauthTokenSecret"]
        assertEquals "unique", account.errors["twitterId"]
        assertEquals "unique", account.errors["admin"]

        // ahora se prueba a asociar el mismo administrador
        // a otra cuenta robot distinta
        // debe pasar
        account = new Account(id:0, screenName:"cuentaprueba2",
                oauthToken: "esteesunoauthtokendeprueba2",
                oauthTokenSecret:  "esteesunoauthtokensecretdeprueba2",
                profileImg: "urlImg2" ,
                twitterId: 111112L,
                batchTime: 15,
                admin: existingAdmin,
                lastSinceDateDM: 1328207838750)

        // se comprueba que da error donde se espera
        assertFalse account.validate()
        assertEquals "unique", account.errors["admin"]

        // se intenta introducir una cuenta correcta. ok
        account = new Account(id: 0, screenName: "cuentaprueba3",
                oauthToken: "esteesunoauthtokendeprueba3",
                oauthTokenSecret: "esteesunoauthtokensecretdeprueba3",
                profileImg: "urlImg3" ,
                twitterId: 111113L,
                batchTime: 15,
```

```
                admin: new Admin(screenName: "adminAccount2",
                password: "jjjjj", mail: "mail2@gmail.com"),
                lastSinceDateDM: 1328207838750)
        assertTrue account.validate()

    }
}
```

Tal y como se puede observar, se comprueba el correcto mapeo de la clase de dominio que representa una cuenta robot. Se provocan errores intencionadamente, tales como: introducir cuentas con datos nulos que deben ser obligatorios, repetir claves únicas, introducir nombres de usuarios de *Twitter* con formatos erróneos, asociar a varias cuentas robots el mismo administrador, etc.

Hay implementada una clase de test por cada clase de dominio:

- `AccountTests.groovy`
- `AdminTests.groovy`
- `UserAccountTests.groovy`
- `AutoSuggestionTests.groovy`
- `OnHashtagTests.groovy`
- `PendingRTTests.groovy`
- `SuggestionTests.groovy`
- `UserAccountTests.groovy`

6.1.2. Otras pruebas unitarias

También se han realizado pruebas unitarias automatizadas para probar las funcionalidades de las clases `DMQueryParser.groovy`, `Mail.groovy` y `TwitterUtil.groovy`.

DMQueryParserTests.groovy

Esta clase valida que los métodos que comprueban los comandos de la aplicación funcionan correctamente. Para ello se introducen diversos casos con comandos con formato erróneo y con comandos con formato correcto.

A continuación se muestra un fragmento de código de esta clase:

```
    @Test
        void validateDMTest() {
        def dev
        // probamos a validar las instrucciones de tipo ADD
        dev = DMQueryParser.validateRequest(
        "ADD -u user1,user2,user3 -q #tag1,#tag2,#tag3")
        assertNotNull dev
        assertEquals dev.get(0), "user1,user2,user3"
        assertEquals dev.get(1), "#tag1,#tag2,#tag3"
        assertEquals dev.get(2), 1
        ...
```

```
        }
```

Tal y como se observa, en ese caso concreto se está probando que el método que valida un comando funciona correctamente en el caso de que se introduzca un comando de tipo `ADD` correcto. Los resultados deben ser tal y como se indican con el `assertEquals`. Si el comando fuese erróneo, el método `validateRequest` debería devolver `null`, por ello se indica con el `assertNotNull` que el resultado esperado no debe ser nulo.

MailTests.groovy

Esta clase valida que los métodos implementados para enviar correos electrónicos funcionan correctamente.

TwitterUtilTests.groovy

Esta clase de prueba valida que la clase que llama a la API REST de *Twitter* funciona correctamente y se recuperan los datos esperados en función de la petición realizada.

6.2. Pruebas funcionales

Las pruebas funcionales llevadas a cabo para comprobar que la aplicación se comporta de la manera deseada han sido las siguientes.

6.2.1. Administrador se da de alta en el sistema

Desde la pestaña *«Alta administrador»*, cuando un usuario se da de alta en el sistema a través de un formulario, se han comprobado los casos que se enumeran a continuación.

1. El usuario no queda registrado en el sistema y se le muestra un mensaje de advertencia indicando el error, que se puede producir en los siguientes casos.
    - Si se introducen campos en blanco.
    - Si se introduce un nombre de administrador ya registrado en el sistema (los nombres de usuario deben ser *case insensitive*, es decir, son indiferentes las mayúsculas de las minúsculas).
    - Si se introduce una dirección de correo duplicada o con formato incorrecto (por ejemplo sin `'@'`).
    - Si se introduce un nombre de usuario que no respete las normas de los nombres de usuario en *Twitter*, que deben tener entre 1 y 15 caracteres alfanuméricos o `'_'`.
    - Si se introduce una contraseña de menos de 5 caracteres.
    - Si se introduce una contraseña que no coincida con la introducida en el campo de validación.

2. El usuario queda registrado en el sistema satisfactoriamente.
    - Si se introducen todos los datos correctamente.

6.2.2. Administrador inicia sesión en el sistema

Desde la opción del menú *«Iniciar sesión administrador»*, cuando el administrador rellena el formulario de inicio de sesión, se han comprobado los casos que se enumeran a continuación.

1. Se produce un error y no se inicia sesión en el sistema. Se le indica al usuario el motivo mediante un mensaje de advertencia.
   - Si se introducen campos en blanco.
   - Si se introduce un nombre de administrador que no esté registrado en el sistema.
   - Si se introduce un nombre de administrador que no corresponda con la contraseña indicada.
2. Se inicia la sesión del administrador en el sistema.
   - Si se introducen las credenciales correctas (nombre de usuario y contraseña). En esta situación pueden darse varios casos.
     - Si es la primera vez que el administrador inicia sesión o si éste no tiene aún asociada una cuenta robot. En este caso el sistema inicia la sesión, y redirige al administrador a una página de *Twitter* en la que se tiene que autenticar con las credenciales de la cuenta de *Twitter* que quiere que sea su cuenta robot asociada. Pueden darse varias situaciones:
       - Desde la página de *Twitter* se declina el acceso de la aplicación a los datos de la cuenta robot. Haciendo esto el administrador no debe tener ninguna cuenta robot asignada y al redirigirle a la interfaz de la aplicación se pide que vuelva a iniciar sesión.
       - Desde la página de *Twitter* se permite el acceso de la aplicación a los datos de la cuenta robot de la que se introduzcan las credenciales, que será una cuenta robot ya asignada a otro administrador. Haciendo esto se muestra un mensaje de error al administrador indicando la situación: no se puede asignar una cuenta robot ya asignada a otro administrador.
       - Desde la página de *Twitter* se permite el acceso de la aplicación a los datos de la cuenta robot de la que se introduzcan las credenciales. Haciendo esto la cuenta robot queda asignada al administrador.
     - Si el administrador ya tiene asignada una cuenta robot, se inicia sesión en el sistema directamente.

6.3. Administrador indica a los usuarios a agregar a su cuenta robot

Iniciada la sesión del administrador, accediendo a la pestaña *«Configuración»* y a la opción *«Ajustes de configuración»*, se han comprobado los casos que se enumeran a continuación al rellenar el formulario con los usuarios y dar al botón *«Enviar»*.

1. No se configuran los usuarios de la cuenta robot debido a un error. Se le indica un mensaje de advertencia al administrador indicando el motivo.
   - Si se deja el campo de usuarios en blanco.
   - Si se introducen más de 1000 usuarios separados por comas.
   - Si se introduce algún usuario que no respete el formato de los nombres de usuario en *Twitter*.
   - Si el nombre de alguno de los usuarios introducidos coincide con el de la cuenta robot.
2. Se asignan correctamente los usuarios agregados a la cuenta robot.
   - Si todos los usuarios introducidos respetan el formato definido.

6.4. Administrador mira la información de configuración de su cuenta robot

Una vez iniciada sesión, configurada la cuenta robot y configurados los usuarios agregados a la misma, el administrador accede a la pestaña de *«Configuración»* y a la opción *«Detalles de configuración»*. Se han comprobado los casos que se enumeran a continuación:

1. Si el administrador ha configurado los usuarios, observar que todos los datos se muestran correctamente y que el intervalo de comprobación calculado automáticamente es el que debe.
    - Si el administrador ha indicado entre (0-25] usuarios, el intervalo de comprobación debe ser 60 minutos.
    - Si el administrador ha indicado entre (25-50] usuarios, el intervalo de comprobación debe ser 30 minutos.
    - Si el administrador ha indicado más de 50 usuarios, el intervalo de comprobación debe ser 10 minutos.

6.5. Ejecutar comandos

Iniciada sesión, configurada la cuenta robot y configurados los usuarios agregados a la misma, el administrador accede a la pestaña de *«Gestión de comandos»* y a la opción *«Ejecutar comandos»*. Se han comprobado los casos que se enumeran a continuación.

1. Se produce un error y se le muestra el mensaje al usuario indicando el motivo.
    - Si el administrador introduce un comando con formato erróneo o deja en blanco la consola y pulsa sobre el botón *«Ejecutar comando»*.
    - Si el administrador introduce un comando con formato correcto pero éste es de tipo `SUGGESTION` (no está permitido ejecutar este comando desde la interfaz).
    - Si el administrador pulsa sobre alguno de los botones para forzar peticiones a la API de *Twitter* (*«Forzar actualización»*, *«Forzar RTs pendientes»* o *«Comprobar DMs»*), o si ejecuta un comando de tipo `UPDATE` y no quedan peticiones suficientes a la API en ese instante (síncronas) o el servidor de *Twitter* está caído.
2. Si el usuario pulsa sobre el botón *«Forzar RTs pendientes»* y no hay ningún RT pendiente para la cuenta robot en ese momento, se le indica al usuario la situación mediante un mensaje.
3. Si el usuario pulsa sobre el botón *«Forzar RTs pendientes»* y hay RTs pendientes, se le indica al usuario mediante un mensaje el número de RTs realizados.
4. Si el usuario pulsa sobre el botón de *«Comprobar DMs»*, y si todo va bien, se le indica al usuario que los DMs han sido comprobados.

6.6. Vista de peticiones activas

Iniciada sesión, configurada la cuenta robot y configurados los usuarios agregados a la misma, el administrador accede a la pestaña de *«Gestión de comandos»* y a la opción *«Instrucciones activas»*. Se han comprobado los casos que se enumeran a continuación.

1. Si el administrador no tiene ninguna instrucción activa para la cuenta robot se muestra un mensaje que lo indica.

2. Si hay instrucciones activas, se muestran con el formato correspondiente.

6.7. Vista de sugerencias

Iniciada sesión, configurada la cuenta robot y configurados los usuarios agregados a la misma, el administrador accede a la pestaña de *«Sugerencias»*.

1. Si no hay ni sugerencias de usuarios ni auto-sugerencias del sistema, aparece un mensaje que lo indica.

2. Si hay solamente sugerencias de usuarios, éstas se muestran y se indica que no hay auto-sugerencias aún.

3. Si hay solamente auto-sugerencias del sistema, éstas se muestran y se indica que no hay sugerencias de usuarios aún.

4. Si hay sugerencias de usuarios y auto-sugerencias del sistema, se muestran las cinco más prioritarias de cada tipo.

Si se intentan agregar o ignorar:

1. Se intenta agregar una auto-sugerencia pulsando sobre su enlace de *«Añadir»* correspondiente. Se observa que ésta se agrega como corresponde a las instrucciones activas.

2. Se ignora una auto-sugerencia pulsando sobre su enlace de *«Ignorar»* correspondiente. Se observa que ésta desaparece de la lista.

3. Se intenta agregar una sugerencia que contenga como recomendación a algún usuario que no pertenezca a la comunidad de la cuenta robot. Al pulsar sobre su enlace de *«Añadir»* correspondiente, se observa que la sugerencia se agrega a las instrucciones activas y que el usuario nuevo se añade a los usuarios agregados a la cuenta robot.

4. Se ignora una sugerencia manual pulsando sobre su enlace de *«Ignorar»* correspondiente. Se observa que ésta desaparece de la lista.

5. Se intenta añadir ejecutando el comando `ADD` alguna de las sugerencias recomendadas. Observar que al añadirla manualmente en vez de a través del enlace, ésta también debe desaparecer de las recomendaciones.

6. Hacer que un usuario recomiende seguir un *hashtag* para un usuario que sea la propia cuenta robot. Esta sugerencia debe descartar a dicho usuario y no aparecer en las recomendaciones. Esto no está permitido puesto que la cuenta robot no puede hacerse RT a sí misma.

6.8. Comprobación de obtención de DMs

Las pruebas que se han realizado para probar que se recuperan bien los DMs son:

1. Enviar un DM a la cuenta robot, con un comando de tipo `ADD` o `DELETE` con formato correcto, cuyo remitente no sea su administrador. El mensaje debe descartarse.

2. Enviar un DM a la cuenta robot, con un comando de tipo `SUGGESTION` con formato correcto, cuyo remitente no sea ninguno de los usuarios agregados a la cuenta robot. El mensaje debe descartarse.

3. Enviar DM a la cuenta robot, con un comando de tipo `ADD` o `DELETE` con formato correcto, cuyo remitente sea su administrador. El comando debe procesarse.

4. Enviar DM a la cuenta robot, con un comando de tipo `SUGGESTION` con formato correcto, cuyo remitente sea su administrador. El comando debe procesarse.

5. Enviar DM con formato incorrecto a la cuenta robot cuyo remitente sea su administrador.

6. Enviar varios DMs consecutivos y forzar la comprobación dos veces seguidas. Observar que la primera vez se detectan los DMs nuevos, pero que la segunda no se vuelven a leer los que ya han sido leídos.

7. Tener una cantidad considerable de DMs pendientes de leer, de tal manera que al forzar la comprobación no hay suficientes peticiones para recuperar todos los DMs nuevos. Se informa al usuario de la situación y no se parsea ninguno para recuperarlos todos la próxima vez, cuando haya peticiones suficientes.

8. No hay nuevos DMs en la comprobación.

6.9. Actualización de usuarios

Las pruebas que se han realizado para probar que se actualizan bien los usuarios son:

1. Actualizar varios usuarios que tengan *tweets* relevantes desde la interfaz web mediante el comando `UPDATE`. Observar que los tweets relevantes se almacenan. Pulsar el botón *«Forzar RTs pendientes»* y observar que la cuenta robot hace RT de los mismos y los elimina de la cola de pendientes.

2. Volver a actualizar los usuarios de la prueba anterior. Observar que ya no se vuelven a coger los *tweets* que han sido *retwitteados*.

3. Forzar actualizaciones cuando hay más usuarios de 25 y observar que se cogen los grupos de usuarios que se han definido para cada intervalo de comprobación (por ejemplo, si hay 53 usuarios, sus comprobaciones automáticas se realizarán cada 10 minutos en grupos de 30 usuarios).

4. Incluir entre los usuarios agregados a la cuenta robot uno que no exista. Al actualizar, se ignorará.

5. Incluir entre los usuarios agregados a la cuenta robot uno que tenga sus tweets protegidos. Al actualizar, se ignorará.

6.10. Envío correo electrónico con sugerencias

1. El correo no es enviado a la cuenta de correo del administrador a las 12:00 PM si no hay sugerencias (ni auto-sugerencias).

2. El correo es enviado a la cuenta de correo de un administrador a las 12:00 PM si hay sugerencias (o de usuarios o auto-sugerencias, o ambas).

Capítulo 7

Conclusiones

En este último capítulo se presentan todas las conclusiones obtenidas durante las fases de desarrollo del proyecto. Previamente fue necesario realizar una investigación exhaustiva sobre las tecnologías utilizadas, sobre todo en la parte referente a desarrollo de aplicaciones con el *framework Grails*.

Al tratarse de una aplicación web, en un principio se pensó desarrollar en JEE, utilizando herramientas de mapeo relacional tales como *Hibernate* y *frameworks* de *inyección* de dependencias tales como *Spring* para dotar a la aplicación de un bajo acoplamiento entre objetos. Investigando se llegó a la conclusión de que *Grails* podría resultar más interesante por todas las ventajas que ofrece, y por su marco de trabajo productivo gracias al paradigma de convención sobre configuración y al principio de no repetir código (DRY).

Otra cosa muy importante a tener en cuenta fue la curva de aprendizaje de *Grails*, muy grata para cualquier desarrollador con conocimientos de *Java* y de JEE.

Grails, también ofrece multitud de *plugins* que proporcionan funcionalidades añadidas. En este caso, ha resultado de muchísima utilidad el *plugin* para *Quartz*, que ha permitido programar las tareas requeridas para la actualización asíncrona de una cuenta robot en el sistema.

Además de llegar al objetivo final definido en los requisitos de diseño de la aplicación, se ha buscado aprender nuevas tecnologías de desarrollo *software* que estén de auge en la actualidad.

Por otra parte, se investigó sobre las APIs que ofrece *Twitter* para que terceros desarrollen aplicaciones sobre este servicio. Existen tres: la REST API, la *Streaming* API y la *Search* API, pero para la aplicación que ha sido objeto de estudio la elegida fue la de REST. Ésta permite acceder a multitud de recursos e información, en concreto a todas las operaciones que están disponibles a través de la interfaz web de *Twitter*. La *Streaming* API no hubiese servido, al igual que la *Search* API. La primera solamente permite recuperar datos en tiempo real, y la segunda tampoco era la más adecuada puesto que recupera los *tweets* que cumplen una *query* determinada con una antigüedad de 7 días. Además, se pueden recuperar un máximo de 1500 *tweets*, frente a los 3200 que permite recuperar la API REST.

Hay que comentar también las dificultades que han surgido durante el desarrollo del proyecto. Los problemas más importantes surgen con la lógica de la aplicación, para la que hay que tener en cuenta multitud de casos que se pueden dar: posibles errores por caídas del servidor de *Twitter*, restricción de 350 peticiones/hora a la API REST, saturación de peticiones al servidor, etc.

Con todo lo comentado en capítulos anteriores del presente proyecto se puede observar que se han te-

nido que tomar decisiones para que la aplicación funcione correctamente en casos extremos. En caso contrario, se podrían perder *tweets* relevantes que tengan que ser *retwitteados* por una cuenta robot.

También ha habido que establecer prioridades entre las distintas operaciones que hay que realizar para actualizar una cuenta robot, estudiar la manera más eficiente para recuperar la información requerida de la API de *Twitter* para minimizar el número de peticiones realizadas, etc. Por ejemplo, si son muchos los usuarios agregados a una cuenta robot, se decidió multiplexar las comprobaciones que el sistema hace para ver si éstos han publicado contenidos relevantes. Si esto no se hiciese así, se podrían producir situaciones indeseadas.

Apéndice A

Manual de configuración e instalación

A.1. Manual de instalación para desarrolladores

En esta sección se indicarán las herramientas que se han utilizado para el desarrollo de la aplicación, así como los pasos que se han tenido que seguir para darla de alta en *Twitter*.

Será necesario:

- Instalar el *SpringSource Tool Suite* (STS). El código de la aplicación se ha implementado con *Groovy & Grails* y, por ello, el entorno de desarrollo elegido ha sido el STS, un IDE con una interfaz muy similar a la del *Eclipse* pero adaptado al *framework Grails* y al lenguaje *Groovy*.
- Instalar *Java*, en concreto la última versión de JDK5 o JDK6.
- Instalar la versión 2.0.1 de *Grails* y la versión 1.8 de *Groovy*.
- Instalar *MySQL* para la base de datos relacional de la aplicación.
- Dar de alta en *Twitter* la aplicación. Para hacerlo hay que acceder a la página: `https://dev.twitter.com/apps`.

A.1.1. Herramientas necesarias para desarrollo

Instalación de Java

Antes de instalar el STS se requerirá que en la máquina esté instalado *Java*, en concreto, las últimas versiones del JDK5 o el JDK6.

Una vez descargado `http://www.oracle.com/technetwork/java/javase/downloads/index.html`, se deberán seguir los siguientes pasos para su instalación.

1. Hacer doble clic sobre el ejecutable descargado.
2. Después aparecerá un cuadro de licencia. Revisar y aceptar los términos de licencia y pulsar sobre el botón *«Accept»* (ver figura A.1).

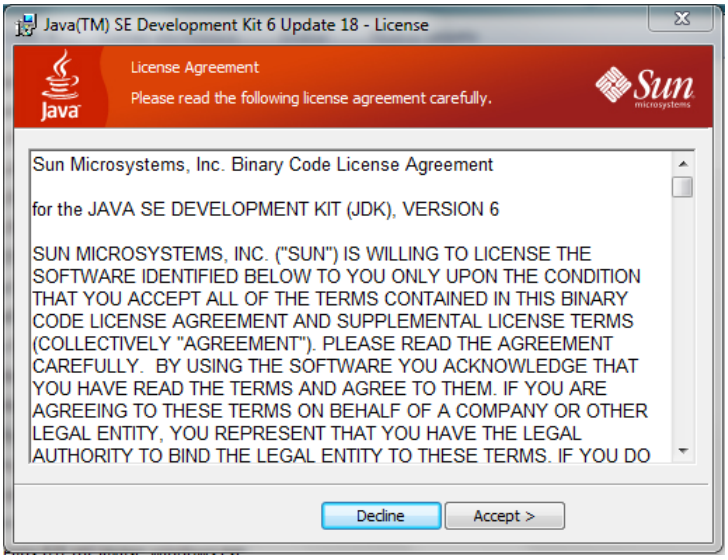


Figura A.1: Instalador JDK: Pantalla 1

3. El asistente mostrará un explorador para que se indique la ruta de instalación. Si ésta se desea cambiar, se modifica, en caso contrario, se deja la ruta por defecto. Pulsar sobre el botón *«Next»* (ver figura A.2).

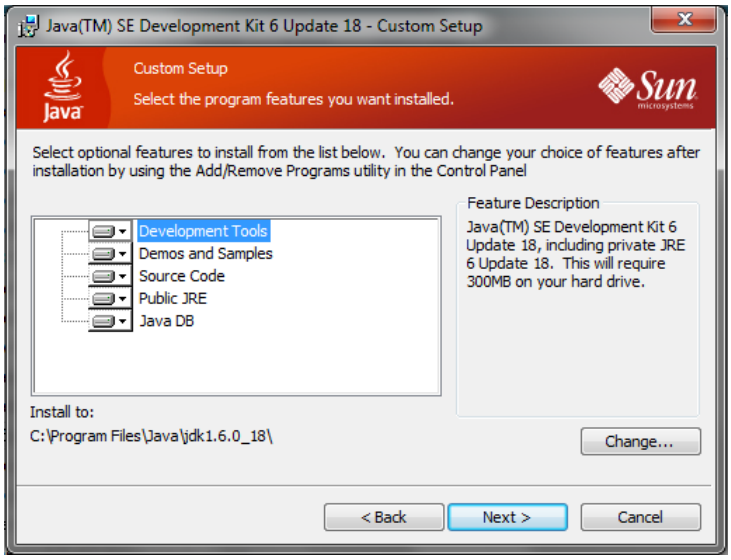


Figura A.2: Instalador JDK: Pantalla 2

4. Posteriormente comenzará la instalación (ver figura A.3). Al terminar, el asistente pedirá que se seleccione la ruta de destino en la que se desea instalar el JRE de *Java*. Se selecciona la ruta deseada (o se deja la que viene por defecto) y se pulsa sobre el botón *«Next»* (ver figura A.4).

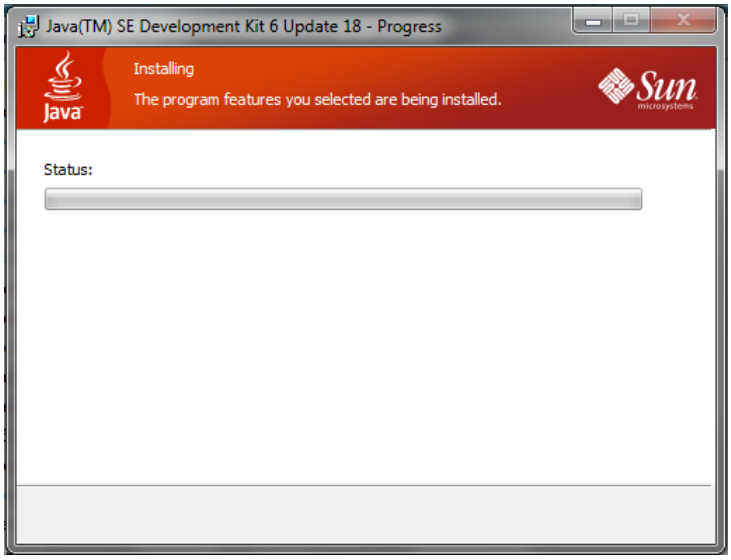


Figura A.3: Instalador JDK: Pantalla 3

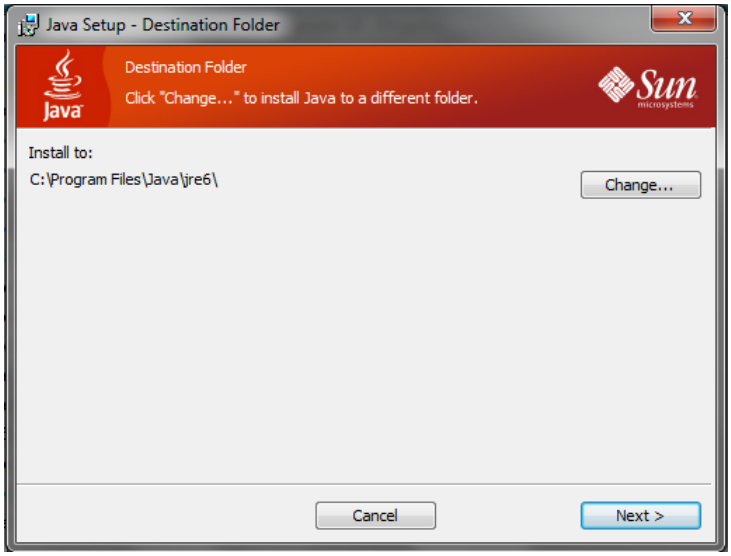


Figura A.4: Instalador JDK: Pantalla 4

5. Nuevamente volverá a aparecer la ventana de progreso de la instalación. Cuando ésta finalice, pulsar sobre el botón *«Finish»*(ver figura A.5).

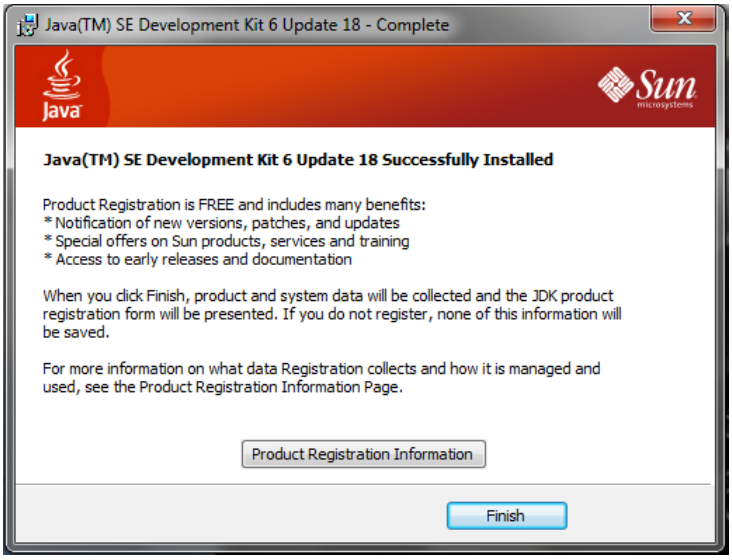


Figura A.5: Instalador JDK: Pantalla 5

Una vez haya terminado el proceso de instalación del JDK y del JRE, se deben configurar las variables de entorno. Para ello:

6. Acceder a las variables del sistema. En el caso de *Windows 7*, pulsar sobre *«Inicio»* ->*«Equipo»* ->*«Propiedades del sistema»*->*«Configuración avanzada del sistema»*->*«Variables de entorno»* (ver figura A.6).

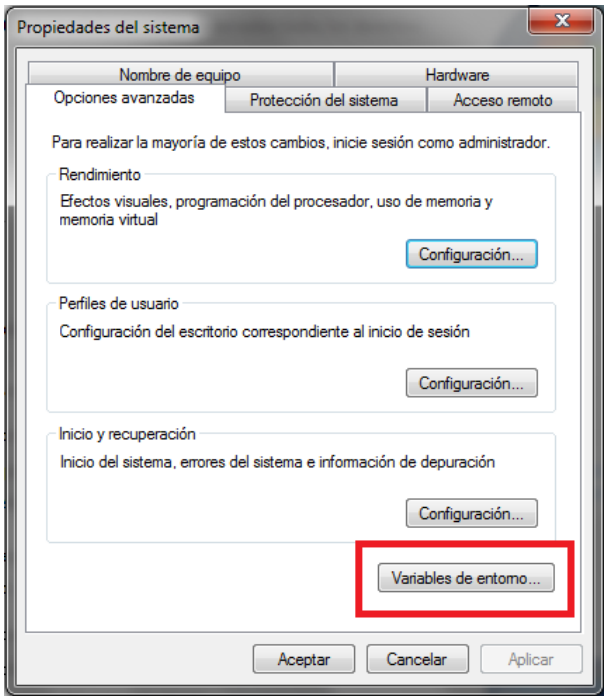


Figura A.6: Instalador JDK: Pantalla 6

7. En las variables del sistema se busca la variable denominada PATH y se edita. Se abrirá una nueva

ventana en la que se deberá agregar la ruta en la que se haya instalado el JDK. En el caso de que se haya dejado la ruta por defecto, ésta será `C:\Program Files\Java\jdk1.6.0_27\bin`, en caso contrario, se deberá escribir la dirección en la que se encuentre instalado el JDK. Para finalizar este paso, se pulsa sobre el botón *«Aceptar»* (ver figuras A.7 y A.8).

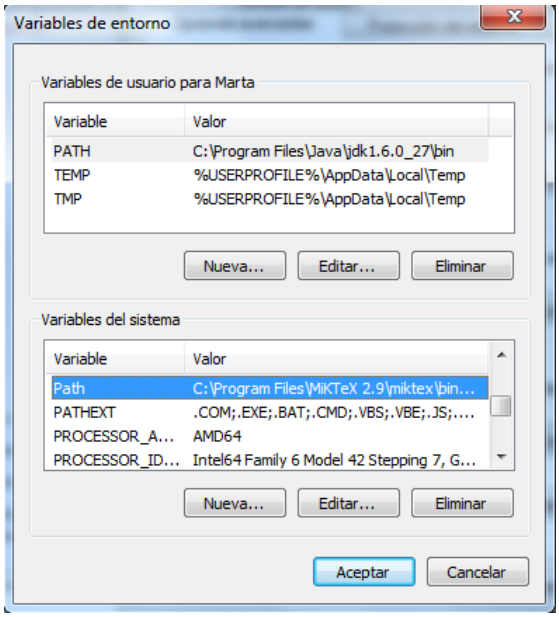


Figura A.7: Instalador JDK: Pantalla 7

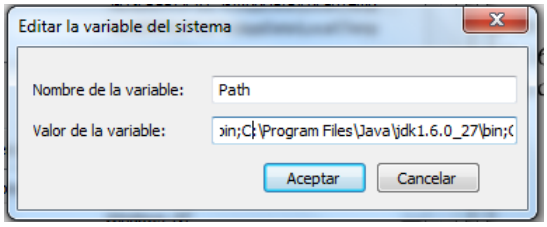


Figura A.8: Instalador JDK: Pantalla 8

8. Por último, se creará una nueva variable de entorno denominada CLASSPATH en la que se definirá la ruta en la que esté localizado el `src.zip`. En el caso del ejemplo, la ruta será `C:\Program Files\Java\jdk1.6.0_27\src.zip`. Para finalizar, se pulsa sobre el botón *«Aceptar»* en todas las ventanas abiertas (la de variables de entorno y la de propiedades del sistema) (ver figura A.9).

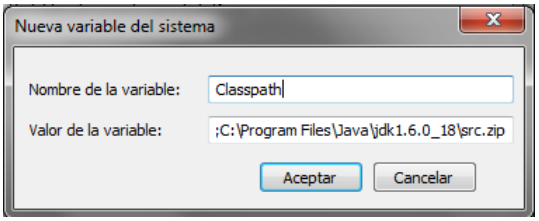


Figura A.9: Instalador JDK: Pantalla 9

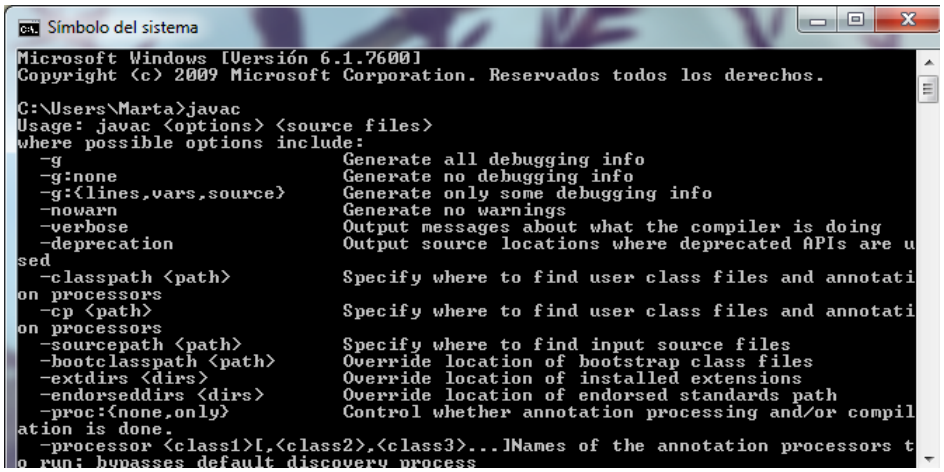


Figura A.10: Símbolo del sistema

9. Como último paso, se puede comprobar que todo ha ido bien abriendo la consola de comandos y escribiendo el comando `javac`. Si *Java* se ha instalado correctamente debería reconocer el comando ejecutado (ver figura A.10).

Instalación del SprinSource Tool Suite

Una vez descargado el STS de la página oficial de *Spring* (`http://www.springsource.com/products/eclipse-downloads`), se debe proceder a su instalación. Para ello hay que seguir los siguientes pasos:

1. Al ejecutar el archivo, se lanzará el asistente. Pulsar sobre el botón *«Next»* (ver figura A.11).

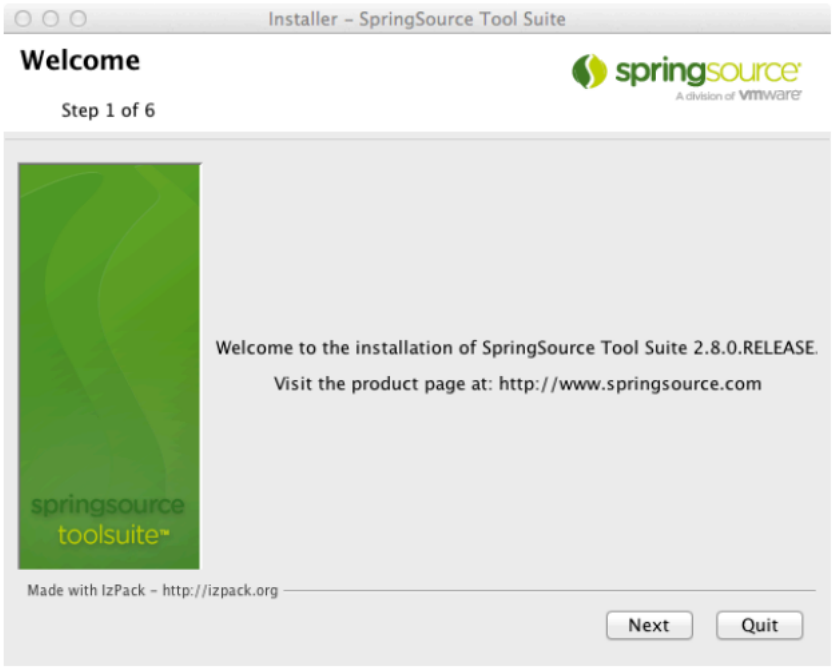


Figura A.11: Asistente de instalación STS

2. Revisar y aceptar los términos de licencia y pulsar sobre el botón *«Next»*.

3. Seleccionar la ruta de instalación y pulsar en *«Next»*.

4. Seleccionar los componentes que se desean instalar y pulsar en *«Next»*.

5. Seleccionar la ruta en la que se encuentre instalado el JDK.

6. A continuación se iniciará el proceso de instalación, una vez que éste finalice, pulsar sobre el botón *«Next»*.

7. Pulsar en *«Next»* hasta que el asistente finalice la instalación.

Para más información consultar la guía oficial de instalación de STS: `http://download.springsource.com/release/STS/doc/STS-installation_instructions.pdf`

Es crucial que, una vez se haya instalado el STS, se instalen también *Groovy & Grails* en la máquina. Este paso se puede realizar fácilmente a través del IDE. Para ello:

1. Abrir el STS e ir al menú *«Help»*->*«Dashboard»*.

2. En el *«Dashboard»*, seleccionar la pestaña *«Extensions»* en la que se verán todas las extensiones que están disponibles para ser instaladas (ver figura A.12).

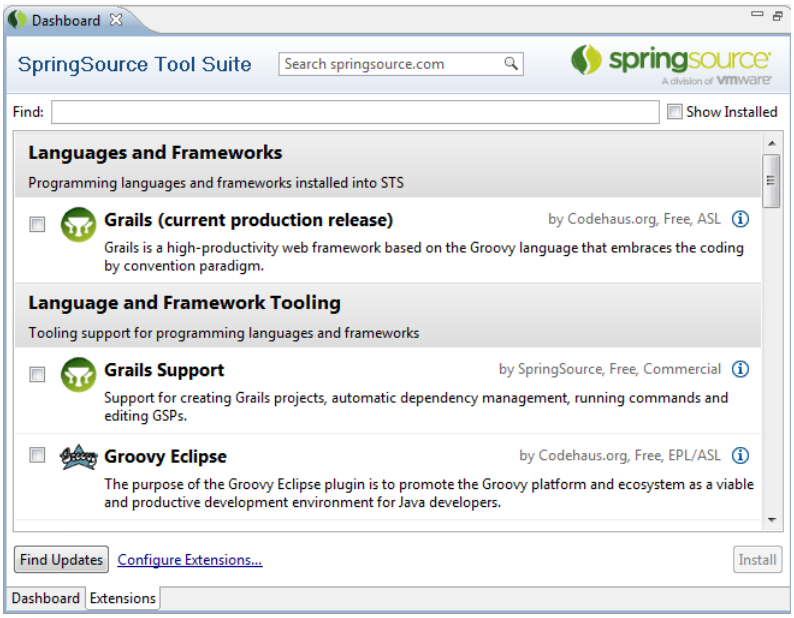


Figura A.12: Instalación extensiones Groovy y Grails 1

3. Seleccionar las extensiones que hacen referencia a *Groovy* y a *Grails* (*Grails current production release*, *Grails support* y *Groovy eclipse*) y pulsar sobre el botón *«Install»*. Después, el STS abrirá una nueva ventana de diálogo con más detalles sobre las extensiones (ver figura A.13 y A.14). Habrá que seleccionar todo antes de continuar con la instalación y pulsar sobre el botón *«Next»*.

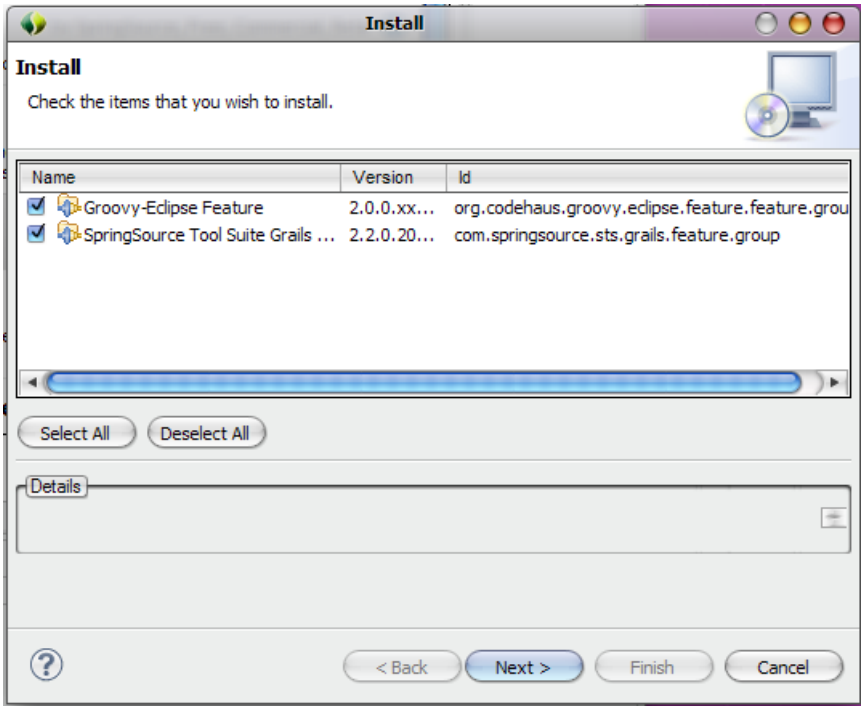


Figura A.13: Instalación extensiones Groovy y Grails 2

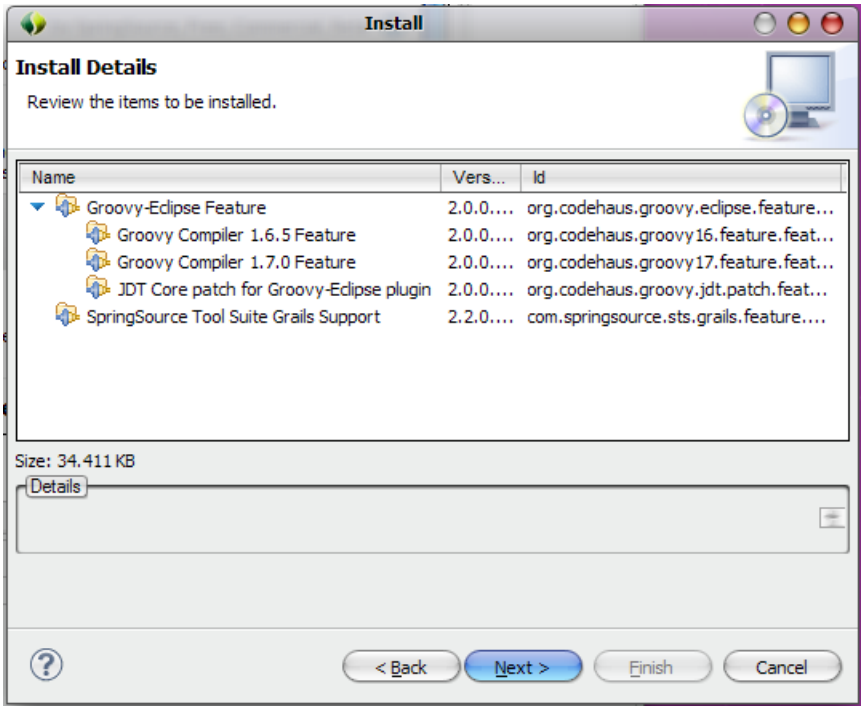


Figura A.14: Instalación extensiones Groovy y Grails 3

4. Aceptar las licencias que se aplican para el soporte de *Groovy & Grails*. Al pulsar sobre el botón *«Finish»* se descargarán todos los archivos necesarios para la instalación de las extensiones (ver figura A.15).

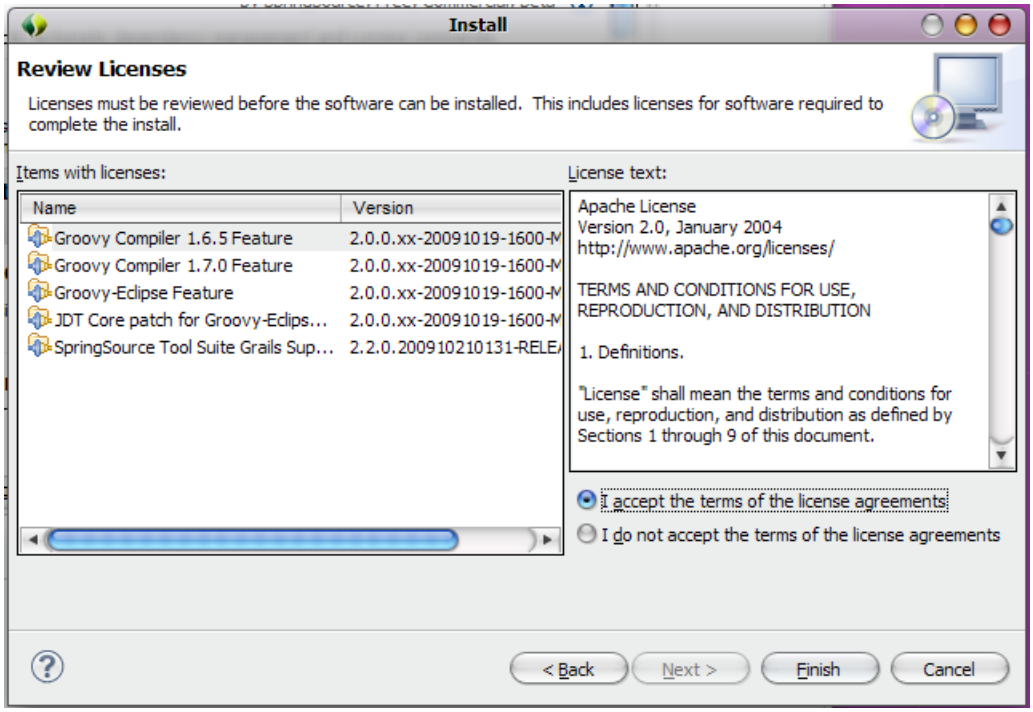


Figura A.15: Instalación extensiones Groovy y Grails 4

Figura A.16: Asistente instalación MySQL

5. Por último, reiniciar el STS.

Instalación de MySQL

Algunos de los datos necesarios para que la aplicación funcione necesitan ser persistidos, por lo que se requiere disponer de una base de datos. Grails trae integrado *HSQLDB*, pero en este caso se ha elegido *MySQL*.

Para instalar *MySQL*:

1. Descargar el ejecutable desde el siguiente enlace `http://dev.mysql.com/downloads/` y hacer doble clic sobre el archivo. Al ejecutarlo, el sistema mostrará un asistente para la instalación del programa. Pulsar *«Next»* (ver figura A.16).

2. Aceptar los términos de licencia. Pulsar *«Next»* (ver figura A.17).

3. Seleccionar la instalación *«Typical»* y volver de nuevo a pulsar sobre botón *«Next»*. (ver figura A.18).

4. Al pulsar sobre *«Install»* comenzará la instalación del programa. Una vez termine, se mostrará una pantalla en la que hay que pulsar *«Finish»* (ver figuras A.19 y A.20).

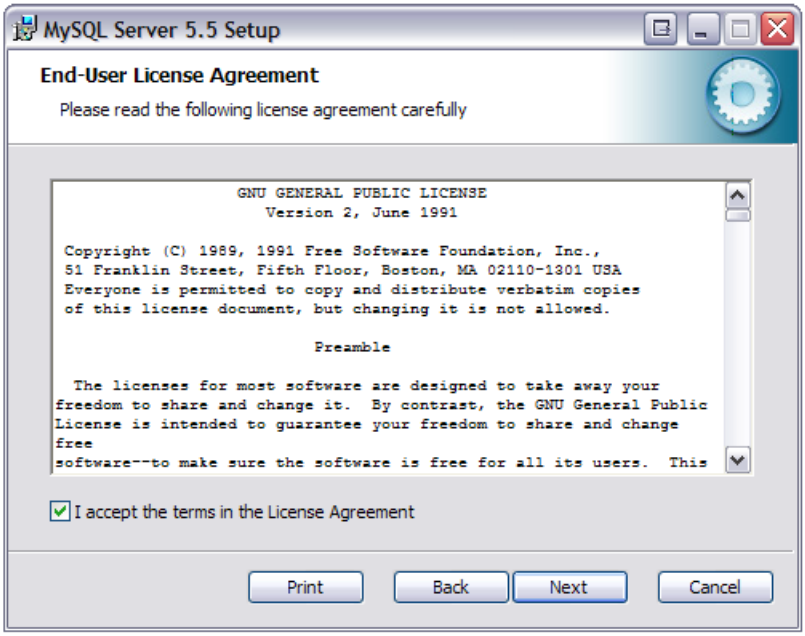


Figura A.17: Asistente instalación MySQL

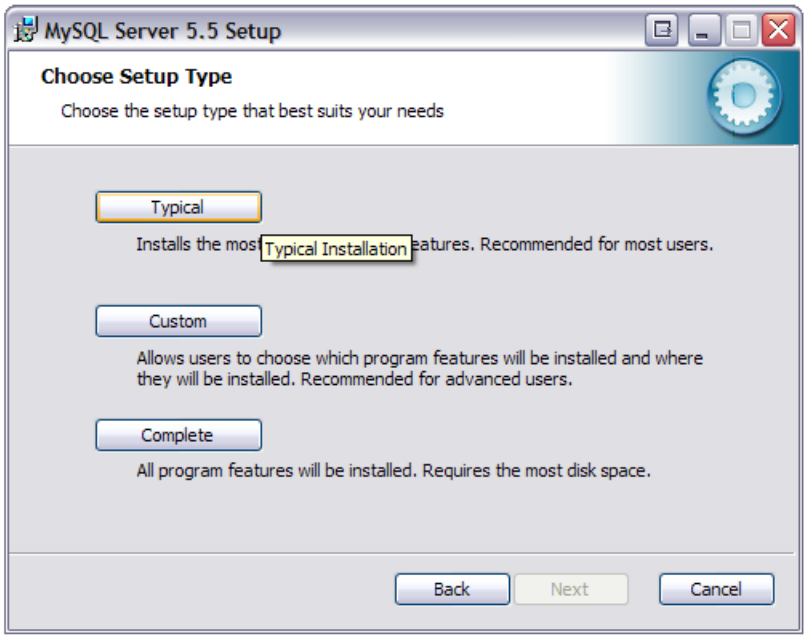


Figura A.18: Asistente instalación MySQL

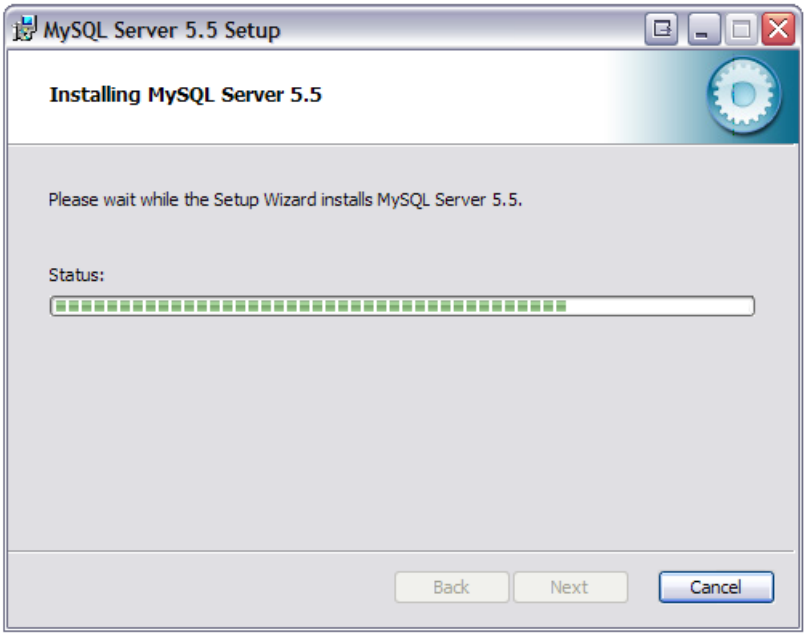


Figura A.19: Asistente instalación MySQL

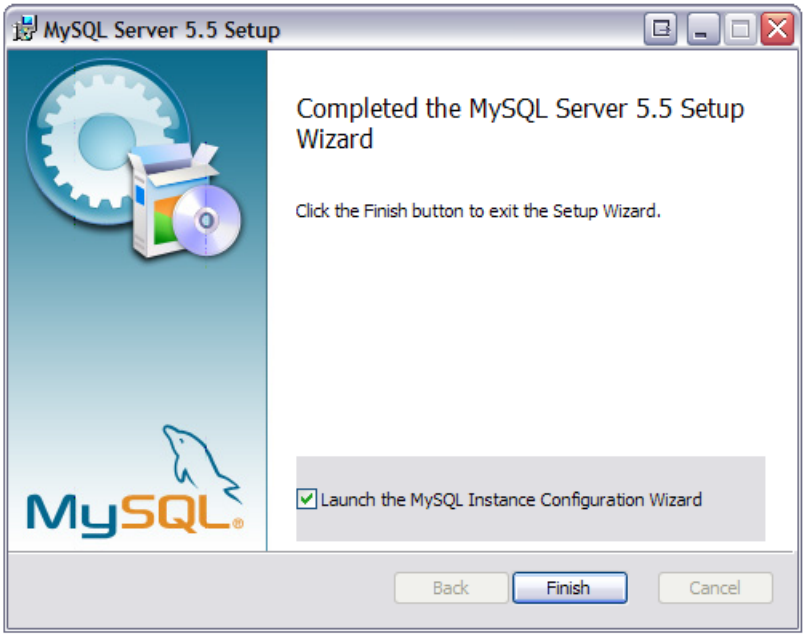


Figura A.20: Asistente instalación MySQL

5. Realizados todos los pasos previos, se abrirá el asistente de configuración del *MySQL Server*. Pulsar sobre *«Next»* (ver figura A.21).

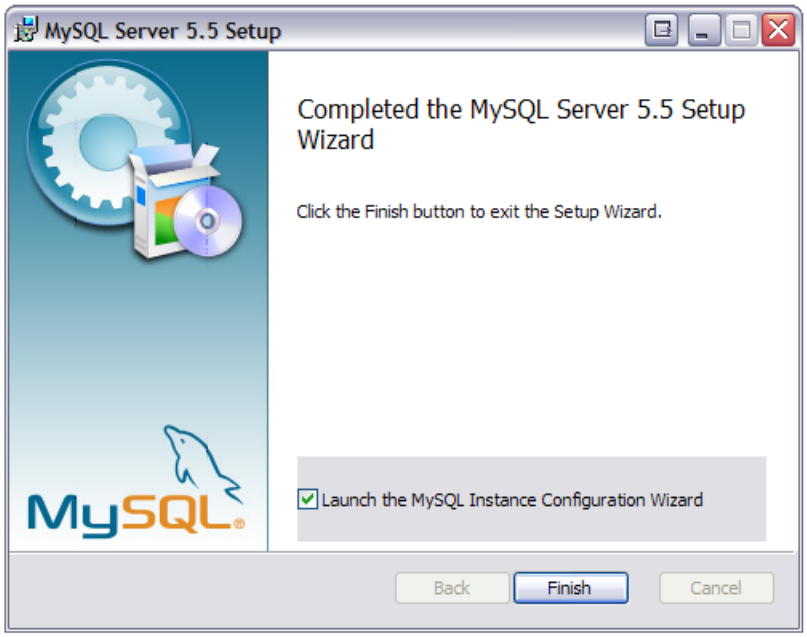


Figura A.21: Asistente configuración MySQL

6. Seleccionar un tipo de configuración y pulsar sobre el botón *«Next»*. En este caso se recomienda la configuración estándar (ver figura A.22).

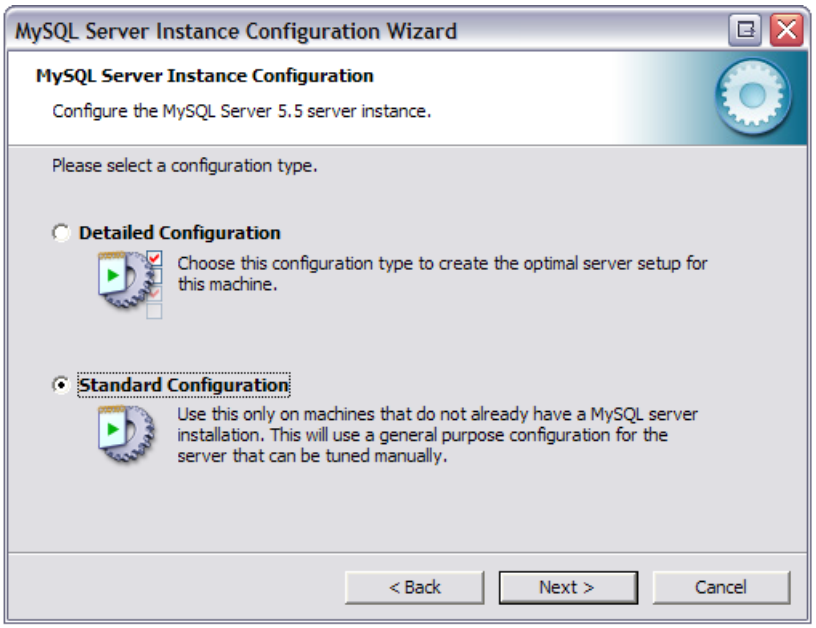


Figura A.22: Asistente instalación MySQL

7. Verificar que están seleccionadas las opciones *«Install as a Windows Service»*, *«Launch the MySQL Server automatically»* e *«Include Bin Directory in Windows PATH»*. Pulsar sobre el botón *«Next»* (ver figura A.23).

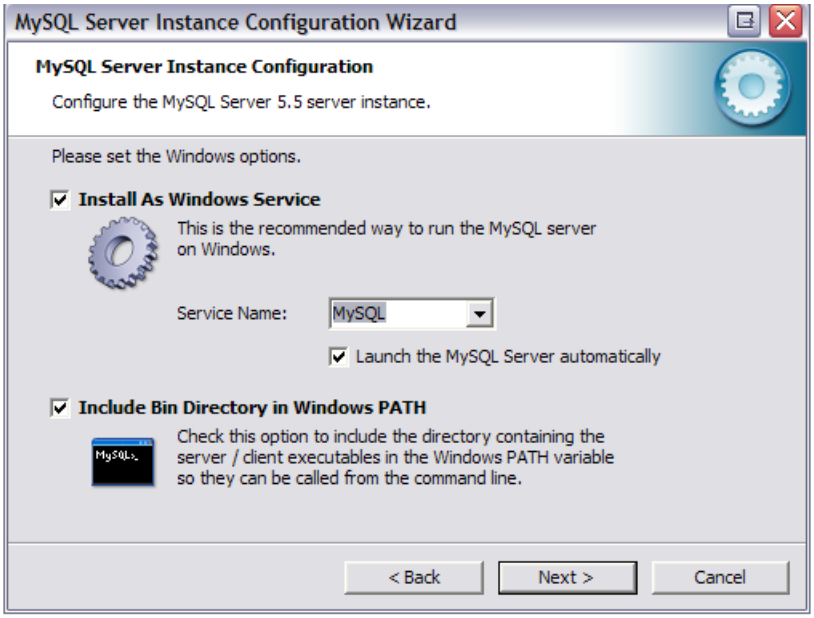


Figura A.23: Asistente instalación MySQL

8. Configurar las opciones de seguridad, ingresando contraseña y nombre de usuario que se desea. Pulsar sobre el botón *«Next»* (ver figura A.24).

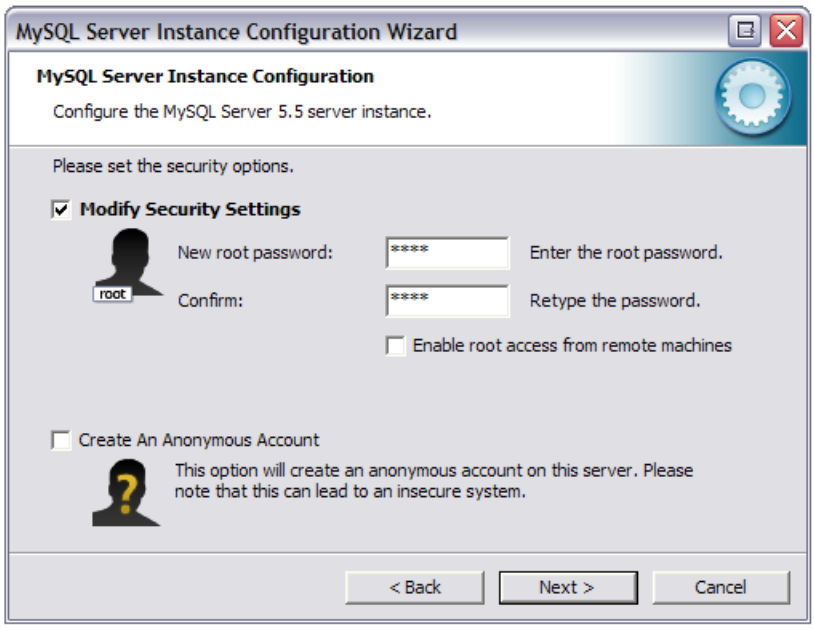


Figura A.24: Asistente instalación MySQL

9. Pulsar sobre *«Execute»* y, tras unos instantes, cuando la intalación finalice, pulsar sobre *«Finish»* (ver figuras A.25 y A.26).

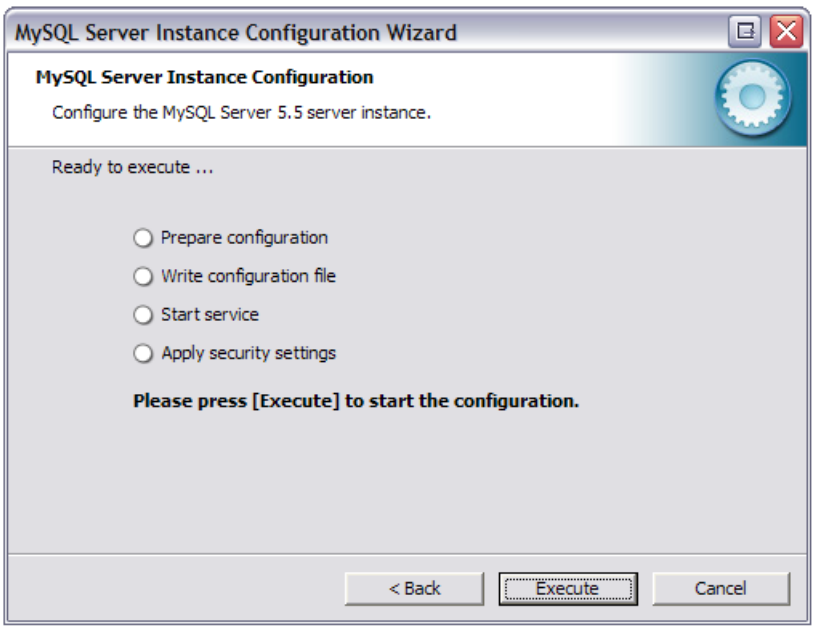


Figura A.25: Asistente instalación MySQL

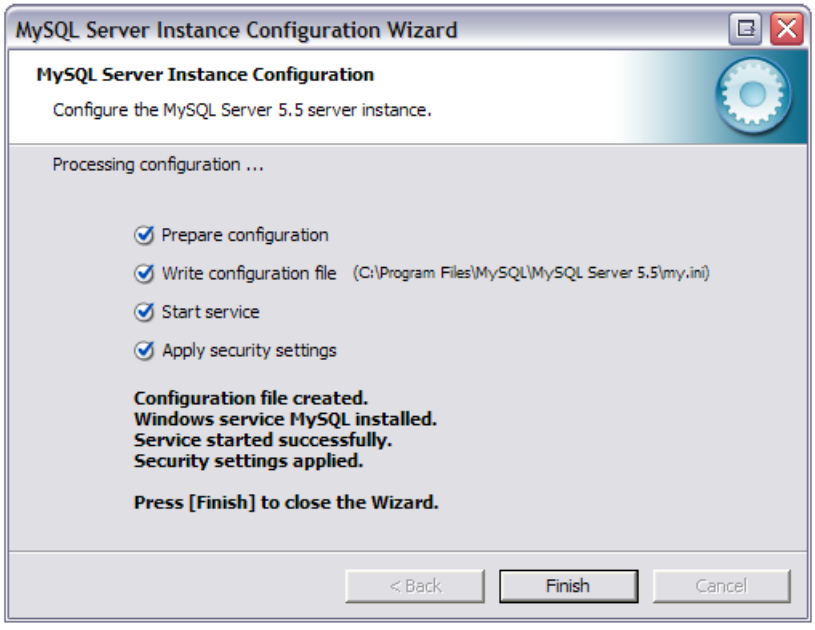


Figura A.26: Asistente instalación MySQL

Figura A.27: Página principal de Twitter

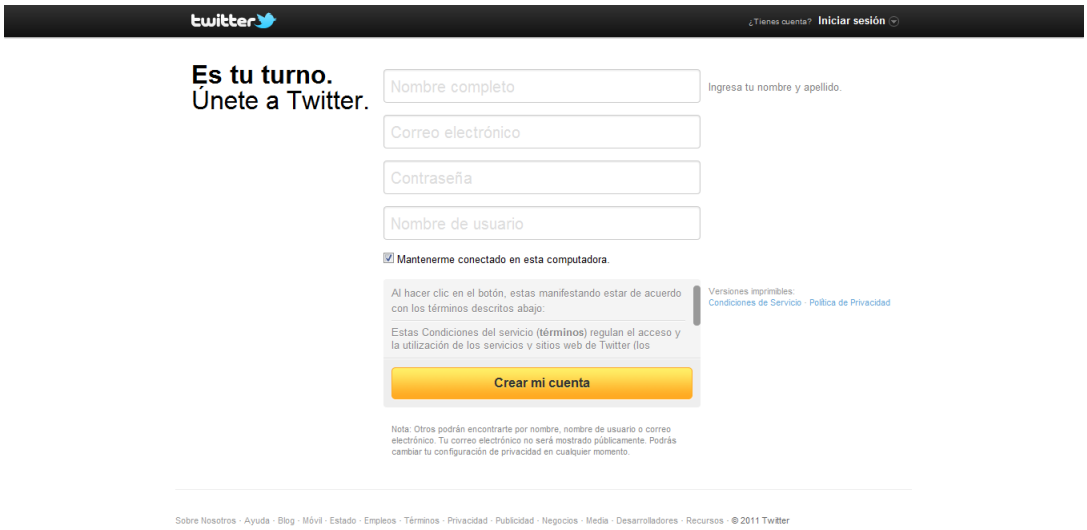


Figura A.28: Formulario de registro en Twitter

A.1.2. Ejecución y puesta a punto de la aplicación

Creación de una cuenta en Twitter

Si no se dispone de ninguna cuenta *Twitter* será necesario registrarse. Esta cuenta se utilizará para dar de alta la aplicación Tuiterbot en *Twitter*. Cómo registrar una aplicación se explicará más detalladamente en el siguiente punto.

Para darse de alta como usuario de *Twitter* hay que seguir los siguientes pasos:

1. Acceder a la página web de *Twitter* a través de la dirección `http://www.twitter.com`, y pulsar sobre el botón *«Regístrate»* (ver figura A.27).

2. Rellenar los datos necesarios que se muestran en el formulario de registro (nombre completo, dirección de correo electrónico, nombre de usuario y contraseña) y, para finalizar, pulsar sobre el botón *«Crear mi cuenta»* (ver figura A.28).

Siguiendo ambos pasos, *Twitter* debería crear la cuenta con éxito.

Dar de alta aplicación en Twitter

Para dar de alta en *Twitter* una aplicación (en este caso Tuiterbot) hay que acceder al siguiente enlace: `https://dev.twitter.com/apps`.

Una vez se haya accedido a la página de registro y el usuario haya iniciado sesión:

1. Pulsar sobre el botón *«Create new application»*.
2. Rellenar el formulario correspondiente (ver figura A.29):
   - *Name*: nombre de la aplicación.
   - *Description*: descripción breve de las funcionalidades de la aplicación.
   - *Web site*: sitio web en el que se encuentra alojada la aplicación o URL desde la que ésta podrá ser descargada.
   - *Callback URL*: es importante que se indique si se trata de una aplicación web, como es el caso. Cuando se proceda a la autenticación mediante *OAuth* de un usuario, *Twitter* redirigirá hacia la *callback* URL.
     En el caso del ejemplo, y suponiendo que la aplicación web va a estar ejecutándose en local, la URL de *callback* que habrá que indicar es `http://127.0.0.1:8080/tuiter-bot/accountManager/callback`. En caso contrario, si la aplicación web se aloja en un servidor externo, la dirección a indicar sería similar, pero cambiando la parte del servidor y el puerto. Expresando la *URL* de un modo genérico, ésta tendría el formato `http://{servidor}:{puerto}/tuiter-bot/accountManager/callback`.
3. Por último, aceptar y pulsar sobre el botón *«Create new application»*. Si todo ha ido bien, *Twitter* redirigirá hacia una página en la que se podrán ver todos los datos de la aplicación, así como el *consumer key* y el *consumer secret* que identificarán a la aplicación de manera única (ver figura A.30).

Nota importante: Para esta aplicación es preciso que se otorguen permisos de lectura, escritura y acceso a mensajes directos. Por defecto, cuando se crea una aplicación en *Twitter*, ésta trae otorgados únicamente permisos de lectura. Para cambiarlos, es necesario ir a la ficha *«Settings»* y en el apartado *«Application type/Access»*, seleccionar *«Read, Write and Access Direct Messages»* (ver figura A.31).

Integración del código en el STS

Una vez se tiene todo lo necesario configurado e instalado en la máquina, se procede a la integración del código fuente en el IDE. En este caso, al tratarse de una aplicación implementada en *Groovy & Grails*, se ha elegido el STS, que tiene un entorno muy similar a *Eclipse*.

Suponiendo que disponemos del código fuente de Tuiterbot y éste se encuentra en una carpeta raíz cuya ruta es `TUITERBOT_HOME`:

1. Abrir el STS.
   a) Seleccionar en el menú: *«File»->«Import»->«Existing projects into workspace»*.
   b) Seleccionar la ruta en la que se encuentra el proyecto descargado y pulsar sobre *«Finish»*.

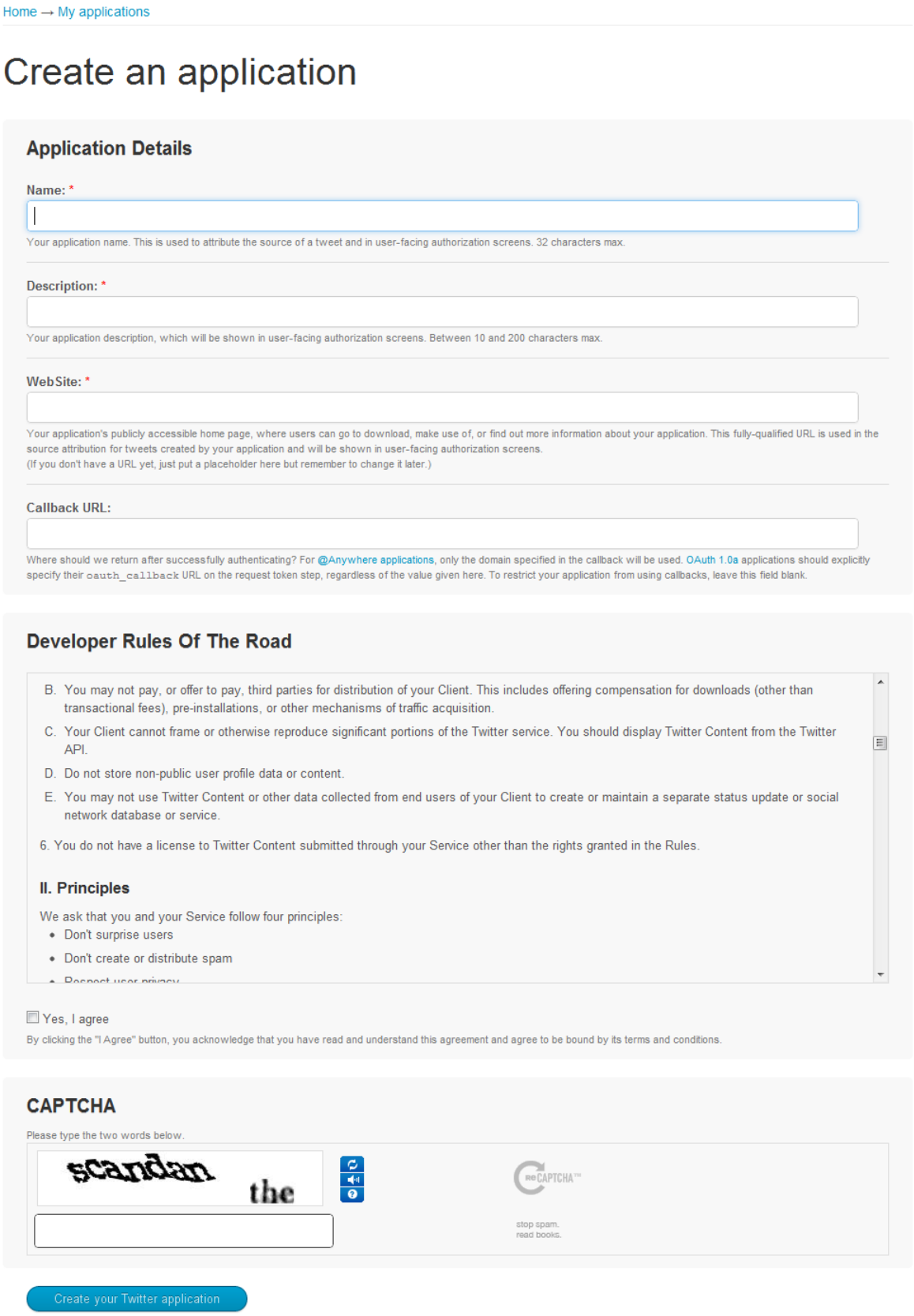


Figura A.29: Registro aplicación en Twitter: Registro aplicación

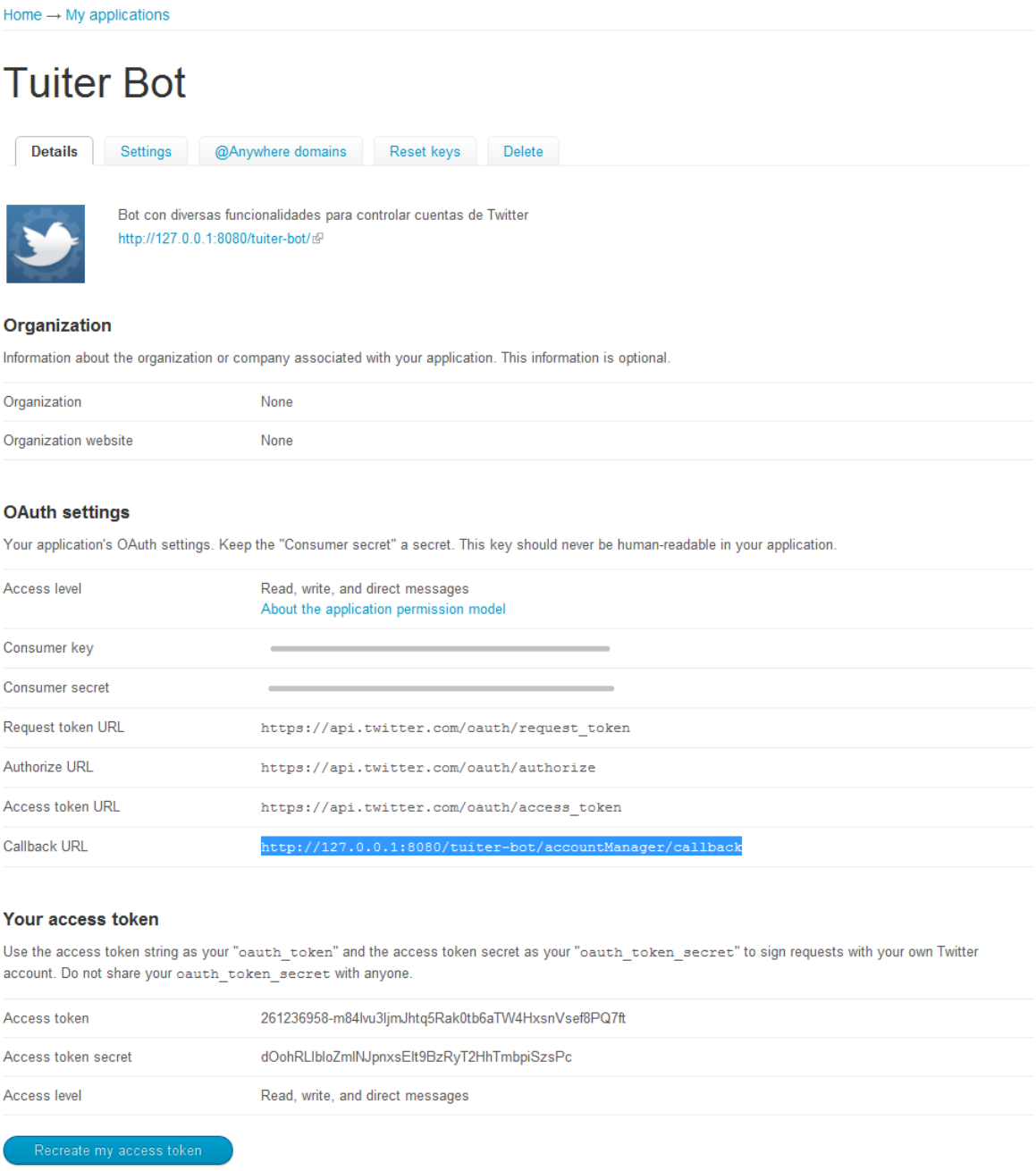


Figura A.30: Registro aplicación en Twitter: Datos aplicación

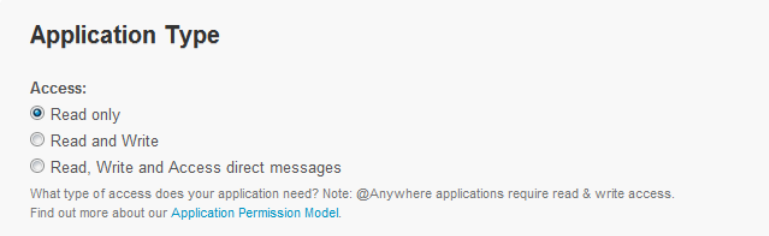


Figura A.31: Permisos de una aplicación Twitter

2. Dirigirse hacia el fichero `TUITERBOT_HOME\grails-app\conf\Config.groovy` tras tener el proyecto importado en el IDE. Cambiar los valores de las propiedades *consumerKey* y *consumerSecret* por los valores correspondientes a la aplicación que se ha debido registrar previamente en *Twitter* (ver figura A.30). En ese mismo fichero, también hay unas propiedades que se pueden modificar concernientes al correo electrónico desde el que la aplicación enviará los correos diarios de sugerencias de seguimiento al administrador de una cuenta robot. En la propiedad `mailSender` se indicará el nombre de usuario de una cuenta de correo de *Gmail* y en `passMailSender`, la contraseña asociada a dicha cuenta.

   Es importante notar que, tal como se ha mencionado anteriormente con la URL de *callback*, se debe configurar de manera correcta la propiedad en la que se hace referencia a la URL del servidor web. Por defecto, ésta apunta al servidor local y al puerto 8080. Si se va a ejecutar la aplicación en un servidor externo, se tiene que cambiar el valor de esta propiedad, denominada `grails.serverURL` (tanto para entornos de producción como para entornos de desarrollo). De manera genérica, el valor que hay que dar a esta propiedad debe seguir el formato `http://{servidor}:{puerto}/${appName}`.

   La parte del fichero de configuración en la que se hace referencia a la URL del servidor es:

```
// set per-environment serverURL stem for creating absolute links
environments {
    production { grails.serverURL = "http://localhost:8080/${appName}" }
    development { grails.serverURL = "http://localhost:8080/${appName}" }
    test { grails.serverURL = "http://localhost:8080/${appName}" }
}
```

   También hay que modificar la propiedad denominada `callbackUrl`, indicando el mismo valor que se haya configurado en (A.1.2).

3. A continuación, se debe poner a punto la base de datos *MySQL*. Para ello:

   a) Primero, acceder al fichero de configuración de la base de datos que se encuentra localizado en `TUITERBOT_HOME\grails-app\conf\DataSource.groovy` y modificar las propiedades `username` y `password`. Estos valores se deben sustituir por el usuario y la contraseña con las que se haya configurado *MySQL*.

   b) Ejecutar el *script MySQL* alojado en `TUITERBOT_HOME\database\mysql.sql`.

   Con esto, ya debería estar todo configurado y listo para poder empezar a usar la aplicación.

4. Por último, desplegar la aplicación en local. Para ello, en el STS, acceder al *Project Explorer*, pulsar con el botón derecho del ratón en la carpeta raíz del proyecto y seleccionar la opción *«run-app»*.

A.2. Manual de instalación para usuarios

En esta sección se indicarán los pasos necesarios para que un usuario instale la aplicación en un ordenador que hará las funciones de servidor web. Para ello es necesario disponer del fichero `tuiter-bot-0.1.war`.

Para poder desplegar la aplicación será necesario instalar previamente:

- *MySQL* para la base de datos relacional de la aplicación.
- *Apache Tomcat 6.x* para poder desplegar la aplicación.

A.2.1. Herramientas necesarias para la ejecución

Para proceder a la instalación de Tuiterbot, habrá que preparar el entorno. Será indispensable tener instalado un servidor web para desplegar la aplicación, y también un sistema de gestión de bases de datos relacionales. Para Tuiterbot se utilizará *Apache Tomcat* como servidor web y *MySQL* como sistema de gestión de bases de datos.

Instalación de MySQL

Ver apartado de instalación de *MySQL* (A.1.1).

Instalación de Apache Tomcat

Para instalar *Apache Tomcat*, previamente es necesario haber instalado *Java* (ver sección A.1.1).
Antes de instalar *Apache Tomcat 6.x*, entrar en la página `http://tomcat.apache.org/` y descargar el `.zip` para *Windows*, ya que el ejemplo de instalación que se explica a continuación está detallado paso por paso para entornos *Windows*, en concreto para *Windows 7* (ni que decir tiene que la aplicación también puede instalarse en entornos *Linux*).

1. Una vez descargado el `.zip`, descomprimir el archivo en el directorio deseado (ver figuras A.32 y A.33).

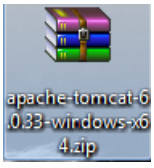


Figura A.32: Icono del .zip descargado de Tomcat

2. A continuación, y si no esta creada ya (si tenemos *Java* instalado seguramente esté ya creada), es necesario crear la variable de entorno JAVA_HOME. Para ello, suponiendo que estamos en un entorno *Windows 7*, acceder a *«Inicio»* ->*«Equipo»* ->*«Propiedades del sistema»* ->*«Configuración avanzada del sistema»* ->Pestaña de *«Opciones avanzadas»* ->*«Variables de entorno»*. En el apartado en el que se visualizan las variables del sistema, pinchar sobre el botón *«Nueva...»* e introducir

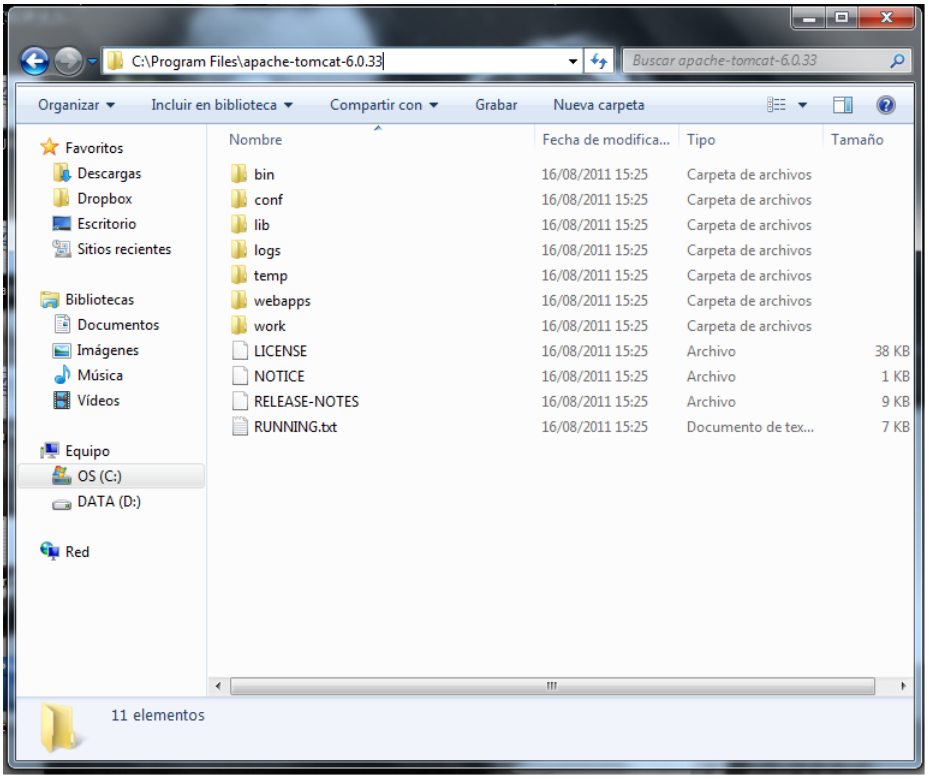


Figura A.33: Contenido del fichero descomprimido

la nueva variable de entorno con nombre JAVA_HOME y valor `C:\Program Files\Java\jdk1.6.0_27`. Pulsar *«Aceptar»*.

3. A partir de ahora se denominará `CATALINA_HOME` al directorio raíz en el que ha sido instalado el *Apache Tomcat*. Dicho esto, existen dos archivos de vital importancia que servirán para iniciar y detener el servidor *Tomcat*. Estos archivos son el `startup` (iniciar) y el `shutdown` (detener), localizados en `CATALINA_HOME\bin` (ver figura A.34). Iniciar *Tomcat* pulsando doble clic sobre el `startup.bat`.

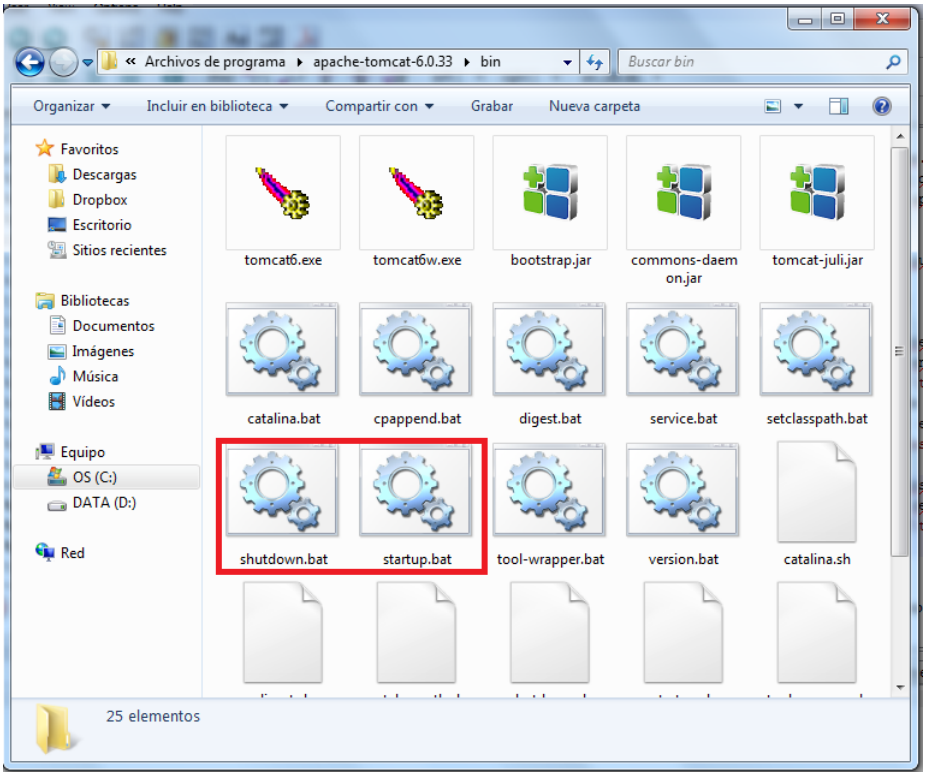


Figura A.34: Ficheros startup y shutdown

4. A continuación, abrir el navegador e introducir la URL `http://localhost:8080/`. Si todo ha ido correctamente, aparecerá la página de inicio de *Tomcat* (ver figura A.35).

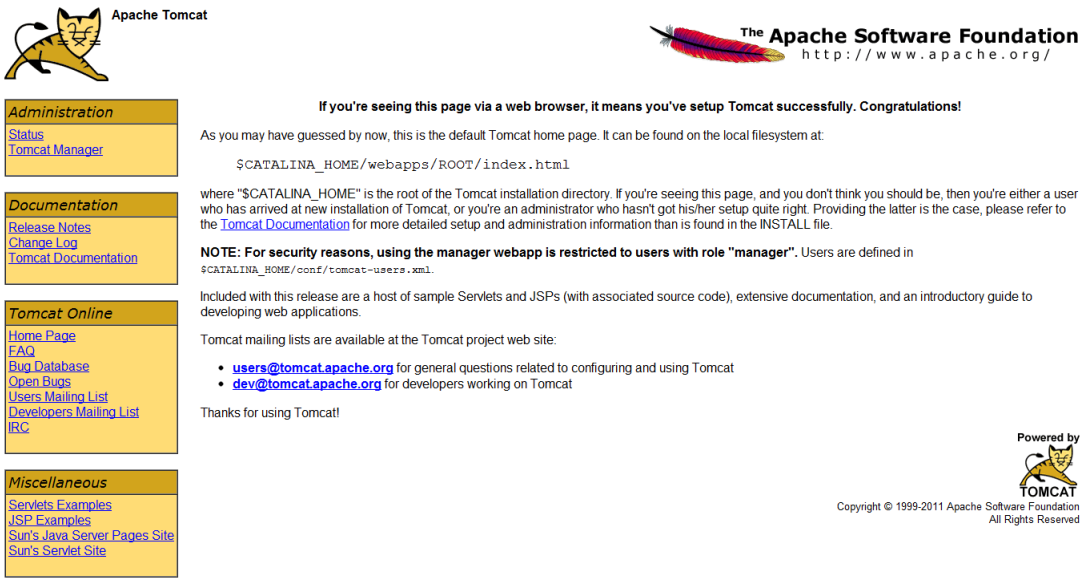


Figura A.35: Página de inicio de Tomcat

5. Para poder acceder a las aplicaciones de gestión y administración de *Apache Tomcat* se necesita

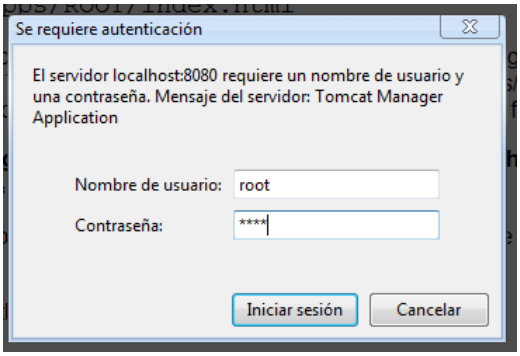


Figura A.36: Acceso a Tomcat manager

crear un usuario. Para ello, acceder al directorio `CATALINA_HOME\conf\tomcat-users.xml` y añadir en el fichero XML la siguiente línea:

(Dentro de la etiqueta `<tomcat-users></tomcat-users>`)
`<role rolename="managergui"/><user username="root" password="root" roles="managergui"/>`

6. A continuación, detener el servidor *Tomcat* y volver a iniciarlo de nuevo. Acceder a la página de inicio de *Tomcat* y pinchar sobre la opción *«Tomcat Manager»*. Aparecerá una ventana emergente para introducir el nombre de usuario y la contraseña. Habrá que introducir los datos del usuario que se había creado en el paso previo (ver figura A.36).

A.2.2. Despliegue y ejecución de la aplicación

Ya instalado y preparado el servidor, ahora solamente queda desplegar el archivo `.war` de la aplicación Tuiterbot. Para ello:

1. Pinchar sobre el botón *«Seleccionar archivo»* que aparece en la parte inferior de la página web del gestor de aplicaciones *Tomcat*. Seleccionar la ruta en la que se encuentre el fichero `tuiter-bot-0.1.war` y pinchar sobre el botón *«Desplegar»* (ver figura A.38).

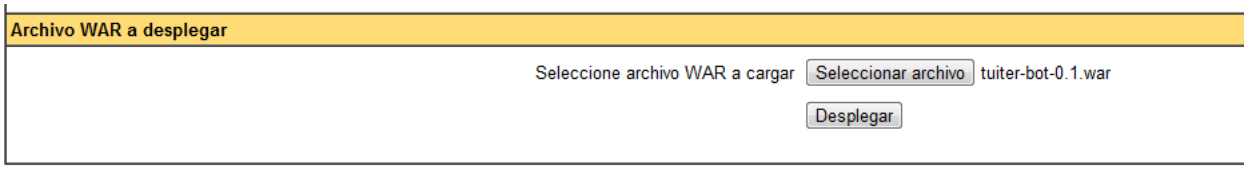


Figura A.38: Desplegar WAR de la aplicación

2. Tras desplegarse la aplicación, ésta aparecerá en el gestor de aplicaciones *Tomcat* y se podrá acceder a su página principal pinchando sobre ella (ver figura A.39).

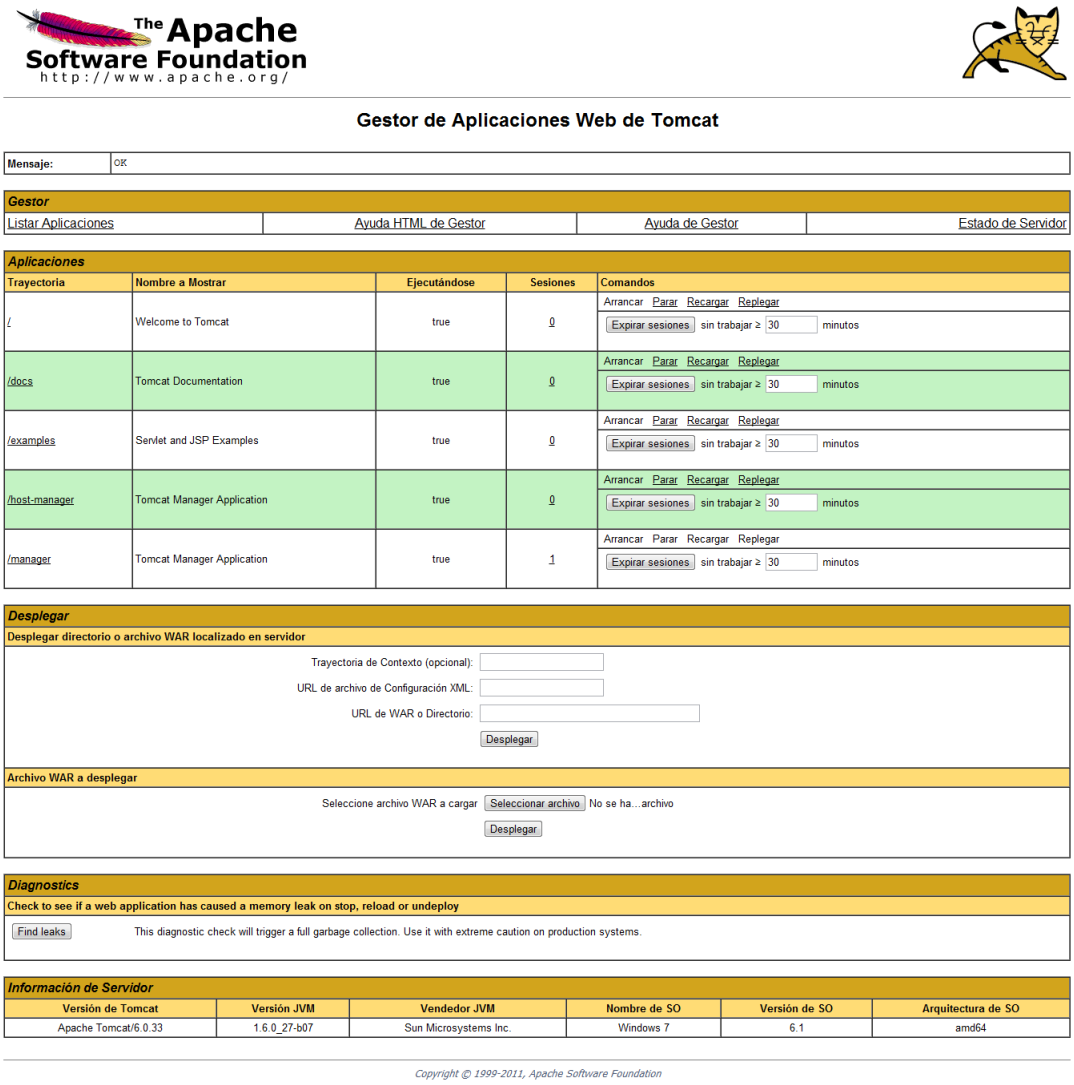
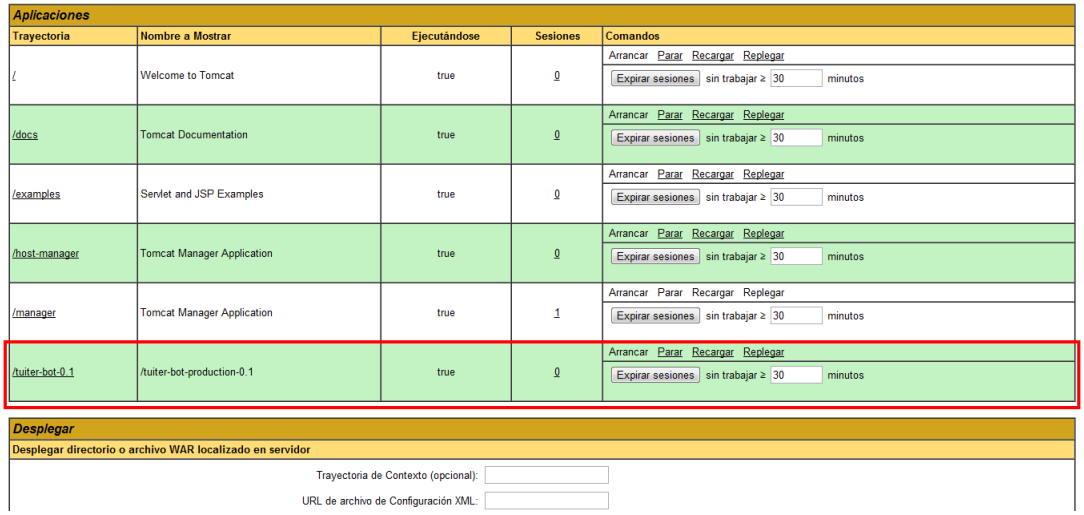


Figura A.37: Gestor aplicaciones Tomcat



Aplicaciones

Trayectoria	Nombre a Mostrar	Ejecutándose	Sesiones	Comandos
/	Welcome to Tomcat	true	0	Arrancar Parar Recargar Replegar Expirar sesiones sin trabajar ≥ 30 minutos
/docs	Tomcat Documentation	true	0	Arrancar Parar Recargar Replegar Expirar sesiones sin trabajar ≥ 30 minutos
/examples	Servlet and JSP Examples	true	0	Arrancar Parar Recargar Replegar Expirar sesiones sin trabajar ≥ 30 minutos
/host-manager	Tomcat Manager Application	true	0	Arrancar Parar Recargar Replegar Expirar sesiones sin trabajar ≥ 30 minutos
/manager	Tomcat Manager Application	true	1	Arrancar Parar Recargar Replegar Expirar sesiones sin trabajar ≥ 30 minutos
/tuiter-bot-0.1	/tuiter-bot-production-0.1	true	0	Arrancar Parar Recargar Replegar Expirar sesiones sin trabajar ≥ 30 minutos

Desplegar

Desplegar directorio o archivo WAR localizado en servidor

Trayectoria de Contexto (opcional):

URL de archivo de Configuración XML:

Figura A.39: Aplicación Tuiterbot desplegada en Tomcat

Apéndice B

Manual de usuario

Si se desea comenzar a usar este servicio, es recomendable leer atentamente todas las instrucciones que se explican a continuación.

B.1. Conceptos previos

Para gestionar una cuenta robot es preciso que el usuario disponga de una cuenta de *Twitter* como mínimo, que será la cuenta robot. Esta cuenta estará gestionada de manera automática por la aplicación, y en ella se publicarán *tweets* de otros usuarios (mediante RT).

Como se detallará en posteriores apartados, para hacer uso de la aplicación, el usuario tendrá que darse de alta previamente en el sistema como administrador. El administrador será el encargado de configurar, a través de la ejecución de una serie de comandos, lo que en la cuenta robot se va a publicar. Para ejecutar comandos un administrador tiene dos opciones:

1. Ejecutarlos de manera síncrona a través de la interfaz web del sistema (estando autenticado).

2. Ejecutarlos asíncronamente mediante el envío de DMs desde su cuenta de *Twitter* a la cuenta robot. El contenido de estos DMs debe ser un comando.

Por tanto, lo ideal es que el nombre de usuario con el que el administrador se da de alta coincida con el de una cuenta de *Twitter*, en caso contrario, no se podría hacer uso de la segunda opción para la ejecución de comandos. Dicha cuenta de *Twitter* puede coincidir o no con el nombre de usuario de *Twitter* de la cuenta robot. Es decir, está permitido que el administrador sea el mismo usuario de *Twitter* que la cuenta robot. Esto se debe a que *Twitter* permite a sus usuarios auto-enviarse DMs.

Durante la lectura de este manual se irán explicando los conceptos con un ejemplo práctico. Para ello se supondrá que un usuario que va a hacer uso de este servicio tiene creadas dos cuentas de *Twitter*. La cuenta robot será `t1000bot`, y el administrador, `cuentaAdmin`.

Antes de detallar cómo configurar una cuenta robot, es importante tener claro el concepto de instrucción activa en el contexto de Tuiterbot.

B.1.1. Instrucción

Las instrucciones representan los contenidos que son relevantes para una cuenta robot y se activan por medio de comandos ejecutados por el administrador. De cómo ejecutar esos comandos y de su formato

se hablará en apartados posteriores.

Una instrucción se identifica por un usuario agregado a una cuenta robot concreta y por un *hashtag*. Ello significa que si el usuario indicado publica un *tweet* que contenga el *hashtag* de la instrucción, la cuenta robot tomará dicho *tweet* como relevante y tendrá que hacer RT del mismo.

Si hay muchos *hashtags* a seguir para un usuario agregado a una cuenta robot, éstos se mostrarán con un formato como el que sigue:

```
usuario ->  hashtags
```

El primer valor es el nombre del usuario, y el segundo valor es una cadena de *hashtags* separados por comas.

Por ejemplo, si se tiene la siguiente configuración:

Cuenta robot: `t1000bot`
Cuenta administradora: `cuentaAdmin`
Usuarios seguidos: `fotogramas_es, cinemania_es, sundancefest`

Suponer que el administrador ha activado:

```
fotogramas_es ->  #globosDeOro2012, #oscars2012
cinemania_es ->  #sundance2012
sundancefest ->  #sundance2012, #goyas2012
```

Esto significa que si `fotogramas_es` publica nuevos *tweets* que contengan el *hahstag* `#globosDeOro2012` o el *hahstag* `#oscars2012`, la cuenta `t1000bot` hará RT de los mismos automáticamente. Lo mismo ocurriría si `cinemania_es` publica algún *tweet* con el *hashtag* `#sundance2012` o si `sundancefest` publica *tweets* con los *hashtags* `#sundance2012` o `#goyas2012`.

Es importante indicar que al activarse un *hashtag* concreto para el usuario indicado, éste empezará a ser relevante para *tweets* que se publiquen en **instantes de tiempo posteriores a la activación de dicha instrucción**.

B.2. Peticiones síncronas y asíncronas

B.2.1. Comprobaciones asíncronas

Para tener actualizada constantemente la cuenta robot hay que tener presente las limitaciones de acceso a la API de *Twitter*, que permite hacer como máximo 350 peticiones/hora.

Hay tres tipos de operaciones de actualización, de más prioritaria a menos prioritaria:

1. **Chequeo, uno por uno, de los usuarios agregados a una cuenta robot**. Periódicamente se debe comprobar si éstos han publicado nuevos *tweets* que contengan *hashtags* relevantes que tengan que ser almacenados para que la cuenta robot haga RT.

   El tiempo entre actualizaciones asíncronas se configura dinámicamente de una manera interna que posibilite optimizar al máximo los usuarios chequeados por intervalo de comprobación. Cuanto

más usuarios siga el robot, más tiempo tardará en hacer un barrido por todos ellos para tener actualizada la cuenta.

Para chequear si los usuarios agregados a una cuenta robot han publicado *tweets* relevantes que haya que almacenar para ser *retwitteados*, si una cuenta robot tiene agregados:

a) Entre 1 y 25 usuarios, éstos se comprobarán en grupos de 25 cada 60 minutos.

b) Entre 26 y 50 usuarios, éstos se comprobarán en grupos de 25 cada 30 minutos.

c) Más de 50 usuarios, éstos se comprobarán en grupos de 30 usuarios cada 10 minutos.

2. **Forzar los RTs pendientes**. Esta operación se llevará a cabo cada vez que un grupo completo de usuarios de una cuenta robot haya sido actualizado. Por ejemplo, si una cuenta robot tiene 75 usuarios agregados, esto implica que se comprueben 30 usuarios cada 10 minutos. Suponiendo que se empiezan a comprobar a las 17.00h, si cada 10 minutos se cogen grupos de 30 usuarios y hay peticiones suficientes a la API de *Twitter*, a las 17:30h ya habrán sido comprobados todos los usuarios agregados a la cuenta robot y, por tanto, se harán los RTs de los *tweets* relevantes encontrados.

   En el periodo que va de las 3:00 AM a las 4:00 AM, si quedan RTs pendientes (*tweets* que se hayan almacenado y de los que la cuenta robot aún no haya podido hacer RT por escasez de peticiones a la API), se publicarán todos.

3. **Chequear si hay nuevos DMs** recibidos para la cuenta robot. Este proceso se realizará cada 6 horas.

   Si hay nuevos DMs, se comprobará si los comandos contenidos son correctos y si han sido enviados por el administrador de la cuenta o por los usuarios (en el caso de que sean sugerencias). Si los comandos respetan el formato, se ejecutarán.

B.2.2. Comprobaciones síncronas

Por hora se reservan 60 peticiones síncronas para que el administrador pueda forzar alguna comprobación que considere que es urgente.

El número de peticiones disponibles síncronamente es visible en todo momento por el administrador de la cuenta robot. En su página de ejecución de comandos aparece un contador con este número (también se puede observar el número de peticiones que quedan en total a la API de *Twitter* en ese momento).

El administrador puede consumir las peticiones síncronas:

- Mediante la ejecución del comando `UPDATE`.
- Al forzar actualización de la cuenta (lo que equivale a un `UPDATE` múltiple).
- Al forzar la comprobación de DMs.
- Al forzar los RTs pendientes.

El modo en el que funcionan las peticiones síncronas es el mismo en el que funcionan las asíncronas excepto por la ejecución, que en lugar de ser automática es forzada a través de la interfaz web.

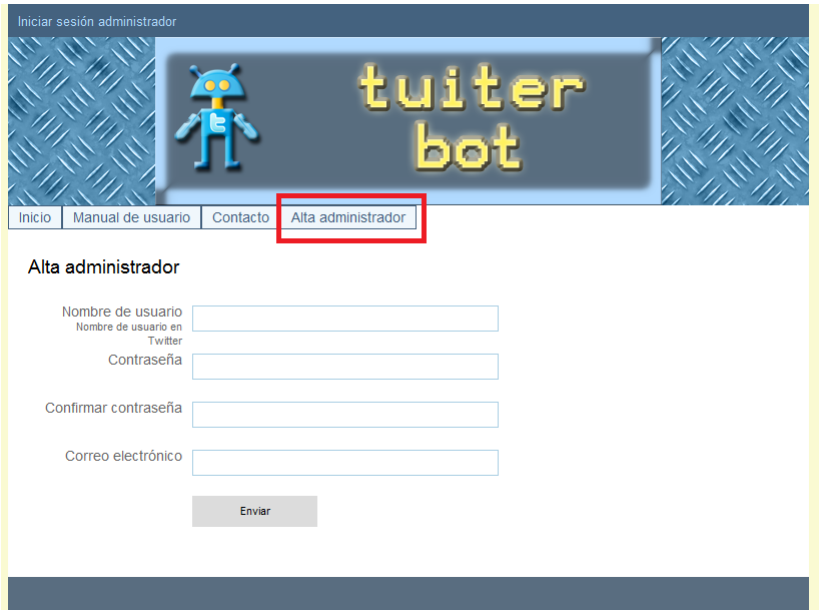


Figura B.1: Formulario alta administrador

B.3. Alta en el sistema como administrador

Para poder usar la aplicación el administrador debe darse de alta en la interfaz web del sistema. Para ello, desde la página inicial de la aplicación:

1. Acceder a la pestaña *«Alta administrador»*. En dicha página aparecerá un formulario de alta (ver figura B.1).

2. Seguidamente, rellenar todos los datos requeridos (nombre de usuario, contraseña y dirección de correo electrónico) y pulsar sobre el botón de *«Enviar»*. Es muy importante que el nombre de usuario sea el mismo que el de la cuenta de *Twitter* que va a ser administradora (en el caso del ejemplo, el valor que habría que introducir es `cuentaAdmin`). .

B.4. Configuración de la cuenta robot

B.4.1. Paso 1. Iniciar sesión en el sistema

Para configurar la cuenta robot el administrador debe autenticarse en el sistema. Para ello debe acceder al enlace *«Iniciar sesión administrador»* que está situado en el menú superior de la interfaz web.

Si inicia sesión por vez primera o si aún no ha asociado ninguna cuenta robot a su cuenta de administrador, éste será redirigido a una página web de *Twitter* en la que se pedirá que introduzca las credenciales de la cuenta de *Twitter* que vaya a ser la cuenta robot (en el caso del ejemplo, `t1000bot`). Si el administrador introduce los datos de la cuenta robot y pincha sobre el botón *«Autorizar aplicación»*, estará dando permisos a Tuiterbot para que tenga acceso a toda su información en *Twitter* (la aplicación podrá acceder a los *tweets* de su cronología, ver a quién sigue y seguir a nuevas personas, actualizar el perfil, publicar *tweets* y acceder a sus DMs) (ver figura B.3). Una vez dados estos permisos, el sistema redirigirá de nuevo al administrador a la página inicial de la aplicación con su sesión ya iniciada (ver figura B.4).

B.4.2. Paso 2. Configurar usuarios de la cuenta robot

Una vez el administrador haya iniciado sesión en el sistema y tenga asociada una cuenta robot, éste debe agregar los usuarios que quiere que sean seguidos por la cuenta robot. Para ello, hay que acceder a la pestaña *«Configuración»* (ver figura B.5) y pulsar sobre el enlace *«Ajustes de configuración»*. Aparecerá un formulario en el que hay un único campo editable, el de los usuarios. Allí se deben introducir todos los nombres de los usuarios de *Twitter* (separados por comas) a los que la cuenta robot va a seguir para comprobar si tienen contenido de relevancia que se deba publicar (ver figura B.6).

En el caso del ejemplo, se tendría que introducir la siguiente cadena de texto en el campo editable de *«Usuarios»*: `fotogramas_es, cinemania_es, sundancefest` y pulsar sobre el botón *«Enviar»*.

En esta pantalla se observa que aparece otro botón denominado *«Confirmar permisos»*. Dicho botón sirve para que, en caso de querer cambiar la cuenta robot asociada a un administrador, se pueda hacer desde la interfaz web. Si se pulsa sobre él, el administrador será redirigido nuevamente a la página de *Twitter* de la figura B.3.

Desde la pestaña de *«Configuración»*, si el usuario pulsa sobre el enlace *«Detalles de configuración»* podrá ver todos los datos relevantes acerca de la configuración de la cuenta robot (ver figura B.7).

- *Cuenta robot*. Es el nombre de usuario de *Twitter* de la cuenta robot.
- *Administrador*. Es el nombre de usuario de la cuenta de *Twitter* del administrador.
- *Usuarios*. Usuarios de *Twitter* agregados a la cuenta robot.
- *Intervalo de comprobación*. Especifica el tiempo entre intervalos de comprobación asíncrona. Esto significa que si el intervalo de comprobación es de 10 minutos, se hará un barrido cada 10 minutos por las cuentas de los usuarios para ver si han publicado nuevos *tweets* de los que la cuenta robot tenga que hacer RT. Este intervalo se calcula dinámicamente en función al número de usuarios a seguir que se introduzcan en la configuración.

B.4.3. Paso 3. Configurar instrucciones de seguimiento

Como se ha mencionado en apartados previos, un administrador puede ejecutar comandos a través de la interfaz web del sistema y mediante el envío de DMs a la cuenta robot desde su cuenta de *Twitter*. Estos comandos permiten añadir instrucciones, modificarlas o eliminar las que no sean de interés, actualizar síncronamente la cuenta robot, etc.

Formato de los comandos del administrador

Existen tres comandos distintos que el administrador puede ejecutar, que son: `ADD`, `DELETE` y `UPDATE`. El último solamente se acepta desde la consola de comandos de la interfaz web (no a través de DMs).

- Comando `ADD`. Este comando se utiliza para añadir instrucciones a la cuenta robot. La sintaxis es la siguiente:
  - `add -u user1, user2..., userN -q #hashtag1, #hashtag2..., #hashtagM`. Significa que se añadirán a la búsqueda todos los *hashtags* indicados en la petición para cada

Figura B.2: Página de inicio

Figura B.3: Autorización de acceso a Tuiterbot

Figura B.4: Pantalla de inicio del administrador

uno de los usuarios especificados.

Por ejemplo, si el administrador de `t1000bot`, `cuentaAdmin`, ejecuta el comando:

```
add -u fotogramas_es, sundancefest -q #oscars2012, #cineDeAutor
```

Para el usuario `fotogramas_es` se activarán los *hashtags* `fotogramas_es ->#oscars2012, #cineDeAutor`, y lo mismo ocurrirá para `sundancefest`, con `sundancefest ->#oscars2012, #cineDeAutor`. Esto significa que si `fotogramas_es` o `sundancefest` publican un *tweet* que contenga alguno de estos dos *hashtags*, se hará RT del mismo en la cuenta robot.

- `add -u * -q #hashtag1, #hashtag2..., #hashtagM`. Significa que se añadirán a todos los usuarios configurados para la cuenta, todos los *hashtags* indicados en el comando.

  Por ejemplo, si se ejecuta el comando:

  ```
  add -u * -q #sundance2012, #goldenglobes2012
  ```

  Suponiendo que la cuenta robot `t1000bot` tiene configurados como usuarios a seguir a `fotogramas_es`, `cinemania_es` y `sundancefest`, se añadirán para cada uno de los usuarios de la cuenta los *hashtags* indicados (`#sundance2012` y `#goldenglobes2012`).

- `add -u user1, user2..., userN -q *`. En este caso ocurre al contrario que en el caso anterior. Se indica que todos los *hashtags* que estén activos en el momento en el que

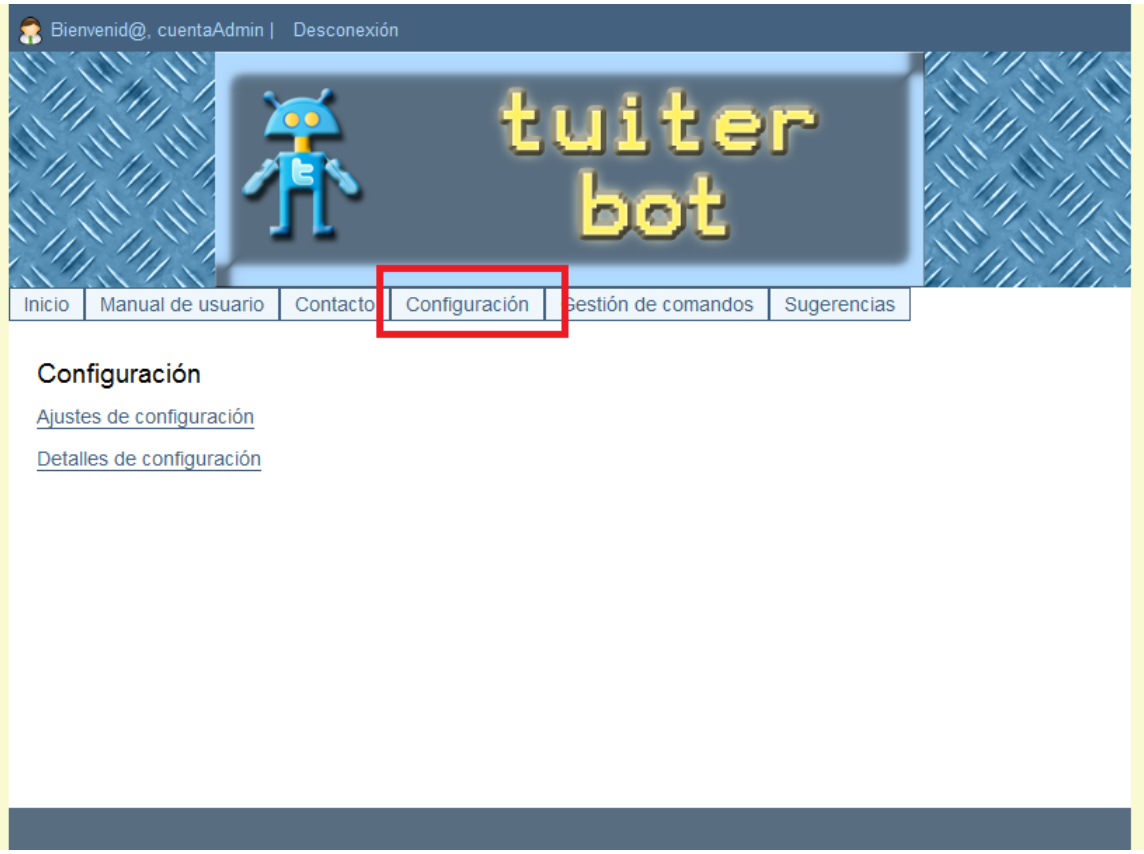


Figura B.5: Menú de configuración

Figura B.6: Formulario de ajustes de configuración

Figura B.7: Detalles de configuración de la cuenta robot

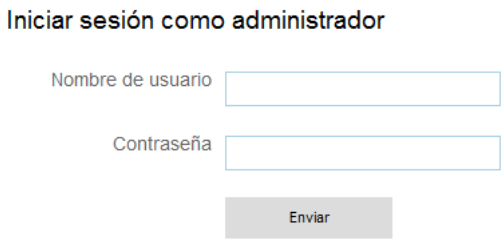


Figura B.8: Formulario de autenticación administrador

se ejecuta el comando (da igual para qué usuarios lo estén), se añadirán a cada uno de los usuarios indicados.

Por ejemplo, si se ejecuta `add -u fotogramas_es -q *`, esto significa que, suponiendo que se han ejecutado las tres instrucciones de los ejemplos anteriores y que, por tanto, los distintos *hashtags* activos para la cuenta robot son `#oscars2012`, `#cineDeAutor`, `#sundance2012` y `#goldenglobes2012`, al ejecutarse el comando, `fotogramas_es` tendrá activos estos cuatro *hashtags*.

- Comando `DELETE`. Tal como el nombre indica, este comando es el que se ejecuta para eliminar instrucciones de las que ya no interese hacer un seguimiento. Bien porque el tema ya se haya quedado obsoleto y no sea de actualidad, o bien porque simplemente ya no interese. La sintaxis es exactamente igual a la del comando anterior pero en vez de añadir instrucciones, las elimina.
  - `delete -u user1, user2..., userN -q #hashtag1, #hashtag2..., #hashtagM`
  - `delete -u * -q #hashtag1, #hashtag2..., #hashtagM`
  - `delete -u user1, user2..., userN -q *`
- Comando `UPDATE`. Este comando permite actualizar todas las instrucciones activas para uno o varios usuarios indicados. Sirve para forzar la actualización que en ese momento interese de manera síncrona si no se desea tener que esperar a que se haga la comprobación automática realizada cada cierto tiempo.
  - `update -u user1, user2..., userN`. Si, por ejemplo, el administrador `cuentaAdmin` ejecuta el comando:

    `update -u fotogramas_es` (suponiendo que `fotogramas_es` tiene activos los *hashtags* `#oscars2012`, `#cineDeAutor`, `#sundance2012` y `#goldenglobes2012`) se hará en ese preciso instante un chequeo de los nuevos *tweets* que haya publicado `fotogramas_es` desde la última comprobación. Si hay algún nuevo *tweet* que contenga alguno de los *hashtags* activos para dicho usuario, se almacenará para ser *retwitteado* por la cuenta robot.
  - `update -u *`. Este formato es lo equivalente a pulsar en el botón *«Forzar actualización»*. Fuerza la actualización en ese mismo instante de todos los usuarios de la cuenta robot. Siempre lo hará en función al número de peticiones que se hayan consumido a la API de *Twitter* y al número de usuarios que haya configurados para esa cuenta robot, ya que si hay muchos usuarios, éstos se multiplexan para comprobar unos cuantos en cada intervalo y evitar que *Twitter* se sature y se agoten las peticiones (tal como se explica en B.2).

Ejecución de comandos desde la interfaz web

Una vez autenticado en el sistema, el administrador debe acceder a la pestaña de *«Gestión de comandos»* (ver figura B.9) y a continuación pulsar sobre el enlace *«Ejecutar comandos»*.

En esta página aparece una consola para escribir, y tres botones, *«Ejecutar comando»*, *«Forzar actualización»* y *«Comprobar DMs»* (ver figura B.10).

- Función del botón *«Ejecutar comando»*. Este botón posibilita la ejecución directa de un comando escrito en la consola.
- Función del botón *«Forzar actualización»*. Si el administrador pulsa sobre este botón, tal y como se ha comentado en la explicación de uno de los formatos del comando `UPDATE`, se forzará la actualización de todos los usuarios que estén configurados para la cuenta robot. Esto se hará total o parcialmente, dependiendo siempre del número de peticiones disponibles en ese instante a la API de *Twitter*, y del número de usuarios que se chequean por intervalo de comprobación.

  Si el grupo de usuarios que se desea chequear es grande y un forzado de actualización no cubre la comprobación completa de todos ellos, siempre que queden peticiones disponibles en ese instante, se puede volver a pinchar sobre el botón las veces que sea necesario para que se comprueben la totalidad de usuarios de la cuenta.
- Función del botón *«Forzar RTs pendientes»*. Internamente, los RTs pendientes se fuerzan cada vez que un grupo entero de usuarios agregados a una cuenta robot se ha terminado de chequear. Mediante este botón el usuario podrá forzar todos los *tweets* que hayan sido almacenados como relevantes hasta ese momento .
- Función del botón *«Comprobar DMs»*. Internamente, los DMs que se reciben en la cuenta robot, se comprueban cada seis horas. Si el administrador desea forzar antes de tiempo el chequeo de estos mensajes, podrá hacerlo mediante este botón.

Volviendo a la pestaña de *«Gestión de comandos»*, si se elige la opción *«Peticiones activas»* el administrador verá una página en la que se mostrarán todas las instrucciones que estén activas para su cuenta robot (ver figura B.11). Cada una de estas instrucciones indica el usuario y el conjunto de *hashtags* que se consideran relevantes para el mismo, tal y como se ha indicado en el apartado B.1.1.

Ejecución de comandos a través de DMs

Como se ha mencionado, los comandos también se pueden ejecutar a través de *Twitter*, mediante el envío de DMs cuyo remitente sea el administrador (`cuentaAdmin`) y cuyo destinatario sea la cuenta robot (`cuentaRobot`). Estos DMs tendrán como contenido alguno de los comandos permitidos para los administradores (`ADD`, `DELETE` o `UPDATE`).

Además del administrador, los usuarios de la cuenta robot también pueden ejecutar un comando de este modo, `SUGGESTION`. En el siguiente apartado se detalla el significado de este comando así como su funcionamiento.

B.5. Sugerencias de seguimiento

Las sugerencias son recomendaciones hechas al administrador por parte de los usuarios de *Twitter* agregados a la cuenta robot o por parte del sistema (que las calcula automáticamente).

Figura B.9: Pestaña gestión de comandos

Figura B.10: Pantalla de ejecución de comandos

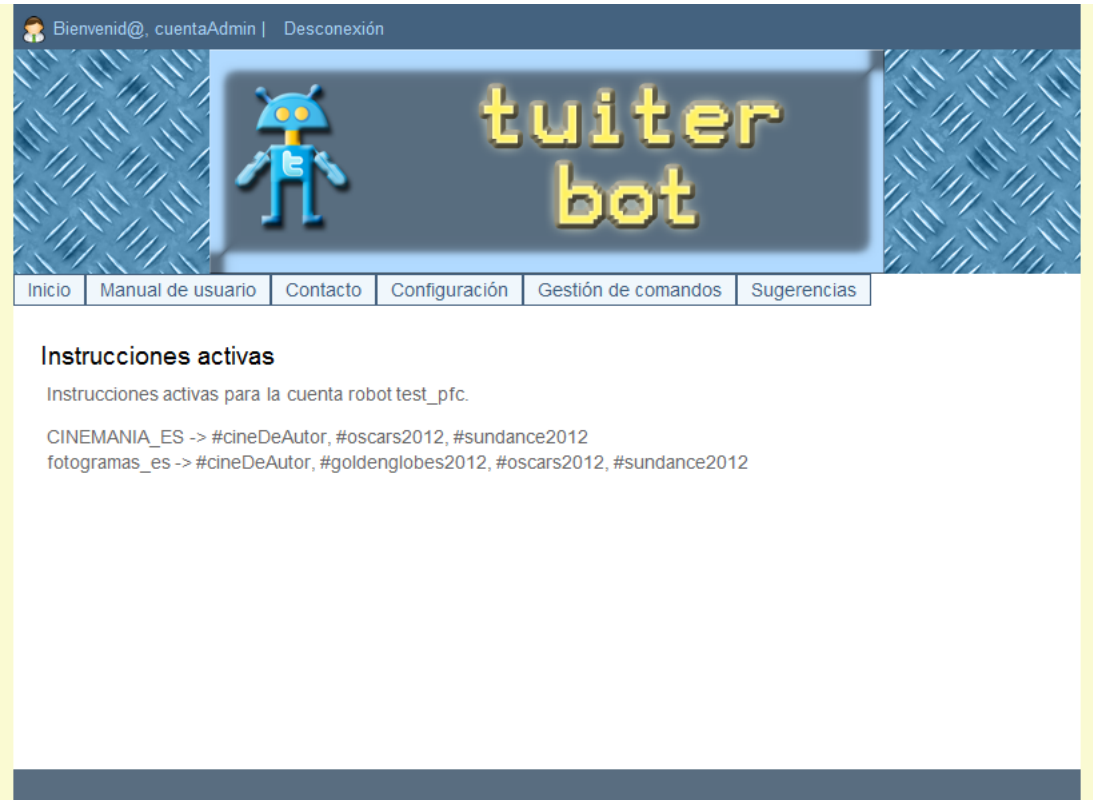


Figura B.11: Pantalla de instrucciones activas

El administrador puede visualizar las sugerencias a través de una pestaña denominada *«Sugerencias»* situada en su interfaz web (ver figura B.12). Esta pantalla muestra como máximo diez sugerencias (las cinco más relevantes enviadas por los usuarios, y las cinco más relevantes calculadas por el sistema). Como se puede observar, junto a cada sugerencia aparecen dos enlaces, uno denominado *«Añadir»* y otro denominado *«Ignorar»*. El primero agrega la sugerencia correspondiente a las instrucciones de seguimiento de la cuenta robot. El segundo ignora la sugerencia en caso de que no resulte de interés para el administrador.

Diariamente, a las 12:00 PM y siempre que haya alguna sugerencia, el administrador será notificado vía correo electrónico de las sugerencias más relevantes que haya en ese momento (serán las mismas que se visualizan desde la pestaña de *«Sugerencias»*).

El correo electrónico tendrá un formato similar al del siguiente ejemplo:

Asunto del correo:
`[Tuiterbot] Sugerencias de seguimiento para t1000bot`

Contenido:
```
Las sugerencias más relevantes enviadas por los usuarios son:
#theExpendables ->  cinemania_es, cinesrenoir, Schwarzenegger, theSlyStallone
#titanic3d ->  cinemania_es, fotogramas_es, imDb
#photos ->  BBCAfrica, natgeotraveler
#newtrailer ->  edmFilmFest, imDb
```

```
#AtlantidaFilmFest ->  filmin

Las sugerencias más relevantes reunidas automáticamente por el sistema son:
#JamesDujardin ->  cinemania_es, cinesrenoir, fotogramas_es, theAcademy,
Uggie_theArtist
#premiosGoya ->  cinemania_es, fotogramas_es, laScript
#sundance ->  sundancefest
#theArtist ->  cinemania_es, fotogramas_es, imDb, Uggie_theArtist
#Miramax ->  sundancefest
```

B.5.1. Sugerencias de los usuarios

Los usuarios que el administrador haya agregado a la cuenta robot podrán enviar sugerencias de seguimiento a la misma.

Siguiendo con el ejemplo de siempre de una cuenta robot con los siguientes datos de configuración:

Cuenta robot: `t1000bot`
Cuenta administradora: `cuentaAdmin`
Usuarios seguidos: `fotogramas_es, cinemania_es, sundancefest`

Suponer que el usuario `fotogramas_es` quiere sugerir a la cuenta robot que siga al usuario `laScript` para los *hashtags* `#berlinale2012` y `#cannes2012`. El usuario `fotogramas_es`, para hacer esta recomendación, tendrá que mandar a la cuenta robot `t1000bot` un DM en el que indique el comando:

```
suggestion -u laScript -q #berlinale2012, #cannes2012
```

Formato del comando SUGGESTION

El formato del comando a ejecutar para recomendar sugerencias es el que sigue:

- `suggestion -u user1, user2..., userN -q #hashtag1, #hashtag2..., #hashtagM`. Para cada uno de los usuarios indicados, se recomienda seguir a cada uno de los *hashtags* indicados.
- `suggestion -u user1, user2..., userN -q *`. Para cada uno de los usuarios indicados, se recomienda seguir todos los *hashtags* que pertenezcan a cualquier usuario agregado a la cuenta robot.
- `suggestion -u * -q #hashtag1, #hashtag2..., #hashtagM`. Para todos los usuarios agregados a la cuenta robot, se recomienda seguir los *hashtags* indicados.

B.5.2. Sugerencias automáticas del sistema

El sistema también puede hacer sugerencias de seguimiento al administrador.

Añadir una de estas sugerencias implica seguir un *hashtag* determinado para alguno de los usuarios de la cuenta robot o para usuarios mencionados en los *tweets* publicados por alguno de los usuarios de la cuenta robot. Esto significa que los usuarios pertenecientes a la comunidad de la cuenta robot publican

tweets que contienen ese *hashtag* con mucha asiduidad.

B.5.3. Ejemplos

En el ejemplo de la figura B.12 aparecen las sugerencias más relevantes de ambos tipos (enviadas por los usuarios de la cuenta robot, y calculadas por el sistema).

Cada sugerencia se compone de un *hashtag* y de un conjunto de usuarios a seguir. Si el administrador pulsa sobre el enlace *«Añadir»*, la sugerencia se añadirá a las instrucciones de seguimiento activas de la cuenta robot.

Retomando el caso de la cuenta robot utilizada en ejemplos anteriores, cuyos datos de configuración son:

Cuenta robot: `t1000bot`
Cuenta administradora: `cuentaAdmin`
Usuarios seguidos: `fotogramas_es, cinemania_es, sundancefest`

Instrucciones activas:

```
fotogramas_es ->  #globosDeOro2012, #oscars2012
cinemania_es ->  #sundance2012
sundancefest ->  #sundance2012, #goyas2012
```

Suponer que el administrador pulsa sobre el *«Añadir»* de la segunda sugerencia más prioritaria de las recomendadas por el sistema. Según la figura B.12, es la sugerencia sobre la que aparece el *tooltip*. Tras añadirse, la configuración de la cuenta robot anterior quedaría como sigue:

Cuenta robot: `t1000bot`
Cuenta administradora: `cuentaAdmin`
Usuarios seguidos: `fotogramas_es, cinemania_es, sundancefest,` **`laScript`**

Instrucciones activas:

```
fotogramas_es ->  #globosDeOro2012, #oscars2012, #premiosGoya
cinemania_es ->  #premiosGoya, #sundance2012
sundancefest ->  #sundance2012, #goyas2012
laScript ->  #premiosGoya
```

Si todo va bien, se habrá añadido el *hashtag* `#premiosGoya` a los usuarios `fotogramas_es` y `cinemania_es`.

Además, como la sugerencia añadida contiene un usuario (`laScript`) no perteneciente a la cuenta robot, se habrá añadido a dicho usuario a la misma, y también se habrá asignado el *hashtag* `#premiosGoya` para dicho usuario.

Una vez añadidas las sugerencias, éstas desaparecerán de la lista. Lo mismo ocurrirá si se rechazan a través del enlace *«Ignorar»*.

Figura B.12: Página de autosugerencias

Apéndice C

Planificación

A continuación se presenta la planificación del proyecto y los costes asociados al mismo.

Para la planificación se ha realizado una división del proyecto en subconjuntos de tareas, cada una de las cuales tiene asociado un tiempo de resolución.

Para el cálculo del presupuesto se ha tenido en cuenta la duración total del mismo y el sueldo estimado para un Ingeniero Técnico de Telecomunicaciones.

El presupuesto será el resultado de la suma de los costes referentes al personal que ha participado en el desarrollo del proyecto, los costes referentes a los materiales empleados y un 20 % adicional del presupuesto de costes indirectos. En estos costes indirectos se incluyen la electricidad, Internet, etc.

Los datos a tener en cuenta para el cálculo de los costes son:

- Sueldo base que se considera para un Ingeniero Técnico de Telecomunicaciones: 62 euros/hora.
- Se consideran 5 días laborables a la semana y jornadas de 8 horas cada día.

C.1. Distribución temporal

Se ha elaborado un diagrama de *Gantt* que muestra el tiempo que se ha empleado para cada una de las tareas enumeradas. Tal y como se puede observar en la figura C.1, el periodo aproximado en el que se ha implementado el proyecto ha sido de unos cinco meses.

Las distintas tareas representadas en el diagrama son las siguientes:

1. Documentación e investigación (15 días).
    a) Búsqueda de información sobre el tipo de aplicación a desarrollar, para hacer un análisis de las tecnologías más convenientes para su desarrollo.
    b) Búsqueda de información sobre las tecnologías escogidas y las herramientas utilizadas.
2. Análisis de requerimientos (5 días).
    a) Enumeración y descripción de la lista de requisitos.
3. Diseño del sistema (5 días).

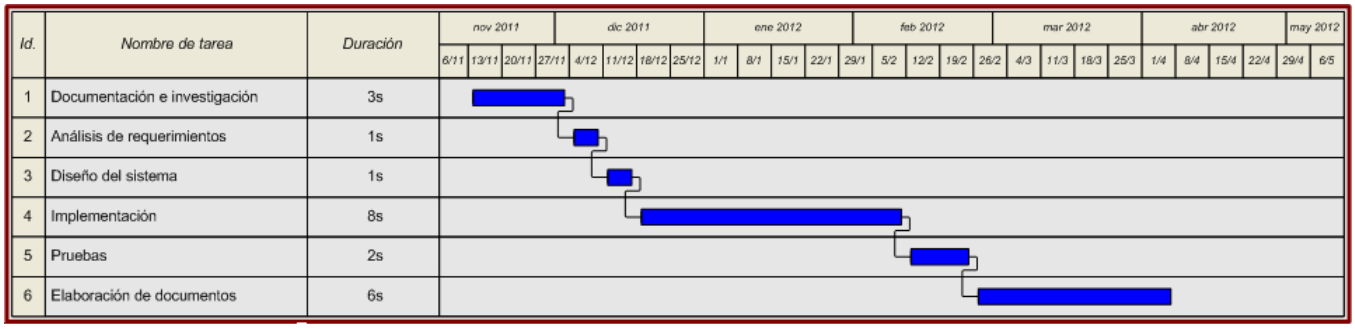


Figura C.1: Diagrama de Gantt

a) Definición de los distintos componentes utilizados.

b) Definición del modelo de datos, control y comunicación.

c) Definición del modelo de la interfaz de usuario.

4. Implementación (40 días).

   a) Implementación de las vistas.

   b) Implementación de las clases de dominio.

   c) Implementación de los controladores y sus servicios correspondientes.

   d) Implementación de otras clases de utilidad.

5. Pruebas (10 días).

   a) Pruebas unitarias y funcionales.

6. Elaboración de documentos (30 días).

   a) Redacción de la memoria de la aplicación.

C.2. Presupuesto

A continuación se presenta el presupuesto final del proyecto desglosado por los distintos tipos de costes que intervienen en el total.

C.2.1. Costes de personal

Los costes asociados al personal se han calculado en función al sueldo base para un Ingeniero Técnico de Telecomunicaciones y al número de horas trabajadas. Se indican en la siguiente tabla:

Tarea	Horas de trabajo
Documentación e investigación	120
Análisis de requerimientos	40
Diseño del sistema	40
Implementación	320
Pruebas	80
Elaboración de documentos	240
Horas totales	840
Coste (euros)	**62 x 840 = 52.080**

C.2.2. Costes de materiales

Los costes asociados a materiales se indican en la tabla que se muestra a continuación. En este caso solamente ha sido necesario un ordenador. Los costes están indicados en euros, y el periodo de depreciación y la dedicación se indican en semanas.

Concepto	Coste	% Uso dedicado	Dedicación	Periodo depreciación	Coste imputable
Portátil Intel	600	100	5	60	49,99

C.2.3. Costes totales

Los costes totales se expresan en la siguiente tabla. Tal y como se ha comentado al principio, a los costes de personal y de materiales hay que sumar unos costes indirectos de un 20 %, en los que se incluyen costes de electricidad, conexión a Internet, etc.

Concepto	Importe (euros)
Coste de personal	52.080
Coste de materiales	49,99
Costes indirectos (20 %)	10.425,998
Total	**62.555,988**

Bibliografía

[1] Wikipedia, artículo *Twitter*: http://es.wikipedia.org/wiki/Twitter [Consulta: 17 de enero de 2011]

[2] Groovy, manual *User guide*: http://groovy.codehaus.org/ [Consulta: 9 de octubre de 2011]

[3] Línea de código, artículo *Tutorial de Groovy*: http://lineadecodigo.com/groovy/tutorial-groovy-instalar-groovy/ [Consulta: 9 de octubre de 2011]

[4] David Marco, artículo *Introducción a Groovy*: http://www.davidmarco.es/blog/entrada.php?id=181 [Consulta: 9 de octubre de 2011]

[5] Dos ideas, artículo *Expresiones regulares en Groovy*: http://www.dosideas.com/cursos/mod/resource/view.php?id=117 [Consulta: 16 de octubre de 2011]

[6] *Manual de desarrollo web con Grails*. Brito, Nacho. Versión 1.0.4. ImaginaWorks Software Factory, 2009. 167 pág.

[7] The Grails Framework - *Reference Documentation*. VV.AA. Versión 2.0.3. SpringSource, 2011. 407 pág. http://grails.org/doc/latest/guide/single.pdf

[8] Wikipedia, artículo *Grails*: http://es.wikipedia.org/wiki/Grails [Consulta: 11 de noviembre de 2011]

[9] Adictos al trabajo, artículo *Planificación de tareas en Java mediante Quartz*: http://www.adictosaltrabajo.com/tutoriales/tutoriales.php?pagina=quartz [Consulta: 10 de noviembre de 2011]

[10] Grails SpringSource, *plugin Quartz*: http://grails.org/plugin/quartz [Consulta: 10 de noviembre de 2011]

[11] Adictos al trabajo, artículo *Expresiones cron*: http://www.adictosaltrabajo.com/tutoriales/tutoriales.php?pagina=expresionesCron [Consulta: 10 de noviembre de 2011]

[12] Twitter, *API REST*: https://dev.twitter.com/docs/api [Consulta: 16 de octubre de 2011]

[13] *OAuth Core 1.0*: http://oauth.net/core/1.0/#anchor9 [Consulta: 10 de noviembre de 2011]

[14] Return (Gis), artículo *Autenticación OAuth*: http://returngis.net/2010/05/autenticacion-oauth/ [Consulta: 10 de noviembre de 2011]

[15] Be code my friend, artículo *Entendiendo OAuth con Twitter*: http://www.becodemyfriend.com/2010/10/entendiendo-oauth-con-twitter-from-scratch/ [Consulta: 10 de noviembre de 2011]

[16] Wikipedia, artículo *CSS*: http://es.wikipedia.org/wiki/Hojas_de_estilo_en_cascada [Consulta: 9 de enero de 2011]

[17] *Guía breve de CSS*: http://www.w3c.es/Divulgacion/GuiasBreves/HojasEstilo [Consulta: 9 de enero de 2011]

[18] Twitter4j, *A Java library for the Twitter API*: http://twitter4j.org/en/index.html [Consulta: 25 de octubre de 2011]